Record

Defining immutable data carriers in a single line in Java.

Intermediate 45 min

Last updated: November 1, 2026
Read online: https://www.learnforgex.com/en/topics/records

Record

In Java, a **record** is a special kind of class designed to carry immutable data, where the compiler writes most of the boilerplate for you. When you write a DTO, a value object, or an API request/response model, instead of hand-writing the constructor, accessors, `equals()`, `hashCode()`, and `toString()`, you define it all in a single line.

What is a Record?

A record is useful anywhere you'd say "this class's only job is to carry a few values together" — a coordinate (`x`, `y`), a monetary amount (`amount`, `currency`), an HTTP response (`status`, `body`), and so on. Writing this kind of class as a regular `class` means hand-writing (or letting the IDE generate) the same five methods — constructor, getters, `equals`, `hashCode`, `toString` — every time; a record delegates this to the compiler.

Why Was It Added?

One of the most frequently criticized aspects of Java was how much repetitive code was required just to write a simple data-carrier class. The same five methods had to be kept manually in sync every time a field was added or changed — if someone forgot to update `equals()`, you'd silently end up with broken comparison logic. Records eliminate this synchronization burden entirely: you define the components once, and everything else is derived from them.

History (Java 14 Preview → Java 16)

Records entered Java 14 as a preview feature under JEP 359, went through a second preview round in Java 15 under JEP 384, and became a permanent, standard language feature in Java 16 under JEP 395. This means Java 21, which this project uses, supports records fully and stably — no preview flag required.

Creating Your First Record

Defining a record is dramatically shorter than the equivalent class. The following single line defines a `Point` record carrying two **components** named `x` and `y`:

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

With this single line, you're telling the compiler: "this type's only job is to carry an `x` and a `y` value together, immutably." In exchange, the compiler generates the following members **for you** (we'll look at each in detail in the "Generated Members" section):

- A **canonical constructor** taking both components as parameters
- Accessor methods named `x()` and `y()`, matching the component names
- An `equals()` that compares all components
- A `hashCode()` consistent with the components
- A readable `toString()` in the form `Point[x=.., y=..]`

Using it is identical to an ordinary class — you construct it with `new` and call its methods:

```
class PointUsage {
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(3, 4);

        System.out.println(p1);                          // Point[x=3, y=4]
        System.out.println("x koordinati: " + p1.x());   // x koordinati: 3
        System.out.println(p1.equals(p2));                // true
        System.out.println(p1 == p2);                      // false — two different objects
    }
}
```

> `x()` and `y()` — note, these are **not** `getX()` / `getY()`. Record accessors use the component name directly rather than the Java Bean convention. This is a deliberate design choice that brings records closer to "data" types in functional languages, and we'll see why it matters in the next section (Record vs Class).

> `record` has been a **keyword** since Java 16, but it is not a *reserved* word — you can still use `record` as a variable or method name (`var record = ...` compiles). Java only interprets it as a contextual keyword at the start of a type definition; this was a deliberate choice to avoid breaking backward compatibility with existing codebases.

Record vs Class

Let's look at how the `Point` example from the "Creating Your First Record" section would look if written as a classic `class`. A `PersonClassic` class carrying the same two fields (`name`, `age`), written immutably by hand, would look like this:

```
final class PersonClassic {

    private final String name;
    private final int age;

    PersonClassic(String name, int age) {
        this.name = name;
        this.age = age;
    }

    String name() {
        return name;
    }

    int age() {
        return age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof PersonClassic other)) return false;
        return age == other.age && name.equals(other.name);
    }

    @Override
    public int hashCode() {
        return java.util.Objects.hash(name, age);
    }

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

The same behavior as a record is a single line:

```
record PersonRecord(String name, int age) {
}
```

Twenty-odd lines of boilerplate collapse into one — but the difference isn't just line count. The record definition gives the compiler extra guarantees:

- A record is **implicitly** `final` — it cannot be extended by another class. Making `PersonClassic` `final` was our own choice (and the right one for immutability); for `PersonRecord`, this is enforced by the language itself.
- A record implicitly extends `java.lang.Record` — just as every enum implicitly extends `java.lang.Enum`. Since Java has single inheritance, this means a record **cannot extend another class** (implementing interfaces is still free — see "Implementing Interfaces").
- The fields corresponding to components are implicitly `private final` — we wrote this by hand in `PersonClassic`; in a record, it's not a choice, it's the rule.
- No setters are **generated** for a record — only accessors (`name()`, `age()`). This is part of guaranteeing immutability.

> Consider adding a third field (say `email`) to `PersonClassic`: you'd need to manually update six places — the field declaration, the constructor parameter, the assignment, the accessor, and both `equals()`, `hashCode()`, and `toString()`. In `PersonRecord`, all you do is add `String email` to the component list; everything else stays automatically in sync. This is exactly where the synchronization burden we mentioned in "Why Was It Added?" disappears.

> Remember that a record cannot be a JPA/Hibernate **entity** — JPA needs a no-args constructor and a non-`final` class to create proxies, both of which go against a record's nature. Records are ideal instead for DTOs, request/response models, and read-only projections (see "Real-World Examples") — entities should still remain ordinary `class`es.

Components

The list inside a record definition's parentheses (`(String name, int age)`) is called the **component list**; each component simultaneously represents a `private final` field, an accessor method, and a parameter in the canonical constructor.

There's no limit on the number of components — even a zero-component record is valid (typically used as a marker or to represent a single event/signal):

```
record Heartbeat() {
}
```

Components can be of any type — primitive, reference type, generic type parameter, or array (we'll see the array trap in the next section, Immutability). Here's a generic record example:

```
record Pair<A, B>(A first, B second) {
}
```

```
Pair<String, Integer> person = new Pair<>("Ada", 30);
System.out.println(person); // Pair[first=Ada, second=30]
```

> An annotation added to a component, if its target is appropriate, is automatically applied by the compiler to the field, the constructor parameter, **and** the accessor method — for example, writing `record User(@NotBlank String username) {}` makes Bean Validation's `@NotBlank` effective on the field, the parameter, and `username()`, even though you only wrote it once.

Generated Members (In Depth)

Let's look one by one at exactly what the generated members we briefly listed in "Creating Your First Record" actually do.

The **canonical constructor** takes all components as parameters, in the order they were defined, and assigns each to the field of the same name — for `Point(int x, int y)`, it does exactly `this.x = x; this.y = y;`, nothing more.

Accessors are generated for each component, named exactly like the component (not with a Java Bean `get` prefix), and directly return the field's value.

`equals()` first checks whether two record instances are of exactly the **same class**, then compares each component in turn. Reference-typed components use `equals()`, primitive-typed components use `==` — **with one exception**: `float` / `double`, which use `Float.compare()` / `Double.compare()` semantics instead of `==`:

```
record Measurement(double value) {
}

class EqualsSemanticsExample {
    public static void main(String[] args) {
        Measurement a = new Measurement(Double.NaN);
        Measurement b = new Measurement(Double.NaN);

        System.out.println(Double.NaN == Double.NaN);   // false — primitive ==
        System.out.println(a.equals(b));                // true  — Double.compare semantics
        System.out.println(a.value() == b.value());      // false — the accessor still
returns a primitive double
    }
}
```

Expected Output

```
false
true
false
```

> This exception matters, because under `Double.compare()` semantics, `NaN` is **equal to itself** (`Double.NaN == Double.NaN` returns `false` with primitive `==`, but `true` inside the record's generated `equals()`) — because a record compares two components with `Double.compare(a, b) == 0`, not a bare `==`. This is the same semantics followed by `Double.equals()` — so the record here is consistent with Java's boxed `Double` behavior, but different from the habitual bare `double == double`.

`hashCode()` combines the hash values of all components (using an unspecified but `equals()`-consistent algorithm) — two equal records always return the same `hashCode()`, which is required for `HashMap`/`HashSet` and similar collections to work correctly.

`toString()` lists the simple class name (without the package prefix) and all components in order, in the form `RecordName[component1=value1, component2=value2]` — this nearly eliminates the need to write a separate `toString()` when debugging.

Immutability

Records being "immutable" is a commonly misunderstood point: what a record guarantees is that its **own references** (its components) can't be changed — final fields, no setters. But if a component holds a reference to a **mutable object**, that object's contents can still be changed from outside the record. This is called "shallow immutability":

```
record Team(java.util.List<String> members) {
}

class TeamMutableTrap {
    public static void main(String[] args) {
        java.util.List<String> names = new java.util.ArrayList<>();
        names.add("Ada");
        names.add("Grace");

        Team team = new Team(names);
        System.out.println(team); // Team[members=[Ada, Grace]]

        // The content changes through the externally held reference, without
        // ever going through Team's own API:
        names.add("Linus");

        System.out.println(team); // Team[members=[Ada, Grace, Linus]] -- the "immutable"
object changed!
    }
}
```

In the example above, the `Team` record itself looks immutable — but the `ArrayList` reference assigned to the `members` field is still held by the calling code, which can change it later; this causes the *content* of the `Team` instance to change without the `Team`'s own API ever being used.

The standard fix is to make a defensive copy inside the **compact constructor** — `List.copyOf()` both copies and makes the result unmodifiable:

```
record Team(java.util.List<String> members) {

    // Compact constructor: no need to repeat the parameter list, just the
    // defensive copy. The assignment (this.members = members;) is done
    // implicitly by the compiler after this block.
    Team {
        members = java.util.List.copyOf(members);
    }
}

class TeamDefensiveCopyUsage {
    public static void main(String[] args) {
        java.util.List<String> names = new java.util.ArrayList<>();
        names.add("Ada");
        names.add("Grace");

        Team team = new Team(names);
        names.add("Linus"); // no longer has any effect on team.members()

        System.out.println(team); // Team[members=[Ada, Grace]]

        try {
            team.members().add("Dennis"); // UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("cannot be modified via members(): " + e.getClass().
getSimpleName());
        }
    }
}
```

Avoid using an **array** as a component. Two reasons: (1) arrays are always mutable, so the same `List` trap above applies to arrays too — and there's no array equivalent of `List.copyOf()`, you'd have to `clone()` manually. (2) more subtly: the generated `equals()` for an array component uses `Object.equals()` (i.e. reference equality), **not** `Arrays.equals()` — if two records hold two *different* arrays with the same elements, `equals()` returns `false`, and `toString()` produces something meaningless like `[I@1b6d3586`. Prefer `List` over arrays almost always; if an array is unavoidable, override `equals()` / `hashCode()` / `toString()` by hand.

`List.copyOf()` throws a `NullPointerException` if the given list contains a `null` element — this lets you use the compact constructor for both defensive copying and an implicit null-element check, in a single line.

Constructors (Canonical, Compact, Validation)

You can also write the canonical constructor we saw in "Generated Members" **by hand** — usually to add validation or normalization. It can be written in two ways.

The **full (explicit) canonical constructor** repeats all the parameters and does the assignments manually — it must have exactly the same signature as the generated one:

```
record Range(int min, int max) {
    Range(int min, int max) {
        if (min > max) {
            throw new IllegalArgumentException("min (" + min + ") cannot be greater than
max (" + max + ")");
        }
        this.min = min;
        this.max = max;
    }
}
```

The **compact constructor** lets you write only the validation/normalization logic, without repeating the parameter list or the assignments — the assignments are done **implicitly** by the compiler at the end of the block:

```
record PersonValidated(String name, int age) {

    // Compact constructor: validation and normalization happen here.
    PersonValidated {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("name must not be blank");
        }
        if (age < 0 || age > 150) {
            throw new IllegalArgumentException("invalid age: " + age);
        }
        name = name.trim(); // reassigning the parameter — not the field
    }
}
```

> Inside a compact constructor, you can reassign a parameter (e.g. `name = name.trim();`) — this is an assignment to the local parameter variable, not yet assigned to the field; the compiler uses this updated value in the implicit assignment after the block. You **cannot** assign directly to the field itself (`this.name = ...`) inside a compact constructor — the compiler rejects this, because the implicit assignment will already be done for you.

A record can define **additional** constructors besides the canonical one — but their first statement must always call the canonical constructor (directly or by chaining) with `this(...)`:

```
record PersonOverloadedConstructor(String name, int age) {

    // Extra constructor: delegates to the canonical constructor with a default
    // age of 0 when the age is unknown. The first line must always be this(...).
    PersonOverloadedConstructor(String name) {
        this(name, 0);
    }
}
```

> This requirement is a deliberate design decision: in an ordinary `class`, a constructor could skip validation and assign fields directly — which could allow some objects to be created in an invalid state. Since **every path** in a record must go through the canonical constructor, the validation you write there **cannot** be bypassed by any way of constructing that record.

Custom Methods

Besides accessors, a record's body can also contain ordinary instance methods, just like a class:

```
record Rectangle(double width, double height) {

    double area() {
        return width * height;
    }

    double perimeter() {
        return 2 * (width + height);
    }

    boolean isSquare() {
        return width == height;
    }
}
```

You can also **override** a generated accessor — for example, to return a defensive copy when exposing a mutable component (an alternative to the `List.copyOf()` pattern from Immutability: copying on read instead of in the constructor):

```
record Snapshot(List<String> items) {
    List<String> items() {
        return List.copyOf(items); // an immutable copy on every call
    }
}
```

> You cannot add an **extra instance field** to a record's body that isn't in the component list — this is a compile error. `record Point(int x, int y) { private int z; }` does not compile. The reason is consistency: a record's "state" consists entirely of its component list, and `equals()` / `hashCode()` / `toString()` are based only on that list — a hidden field these members can't see would break the record's fundamental guarantee. (Static fields are not subject to this restriction — see the next section.)

Static Members

Unlike the other restrictions, a record behaves exactly like an ordinary class when it comes to static fields, methods, and initializer blocks. The most common use is static factory methods and predefined constants:

```
record PointWithFactory(int x, int y) {

    static final PointWithFactory ORIGIN = new PointWithFactory(0, 0);

    static PointWithFactory origin() {
        return ORIGIN;
    }

    static PointWithFactory of(int x, int y) {
        return new PointWithFactory(x, y);
    }
}
```

A factory method like `PointWithFactory.origin()` is more readable than writing `new PointWithFactory(0, 0)` and expresses intent clearly — especially preferred for commonly used "special" values.

Implementing Interfaces

We saw in "Record vs Class" that a record can't extend another class (it already extends `java.lang.Record`) — but just like enums, it can implement as many **interfaces** as you like. One of the most common examples is `Comparable<T>`:

```
record ComparablePointExample(int x, int y) implements Comparable<ComparablePointExample> {

    // Ordering rule: x first, then y if equal (simple lexicographic order).
    @Override
    public int compareTo(ComparablePointExample other) {
        int byX = Integer.compare(x, other.x);
        return byX != 0 ? byX : Integer.compare(y, other.y);
    }
}
```

> The record + interface combination becomes especially powerful when paired with a `sealed interface` — for example, you can define `sealed interface Shape permits Circle, Rectangle {}` and write each subtype as a record. We'll cover this pattern in depth, along with how to use it with modern `switch`, in the **Record Patterns** appendix.

Nested Records

A record can be defined inside another record (or class). Just like enums, a nested record is **implicitly** `static` — it can be used without needing an instance of the enclosing class, since it's simply not possible to define a non-`static` nested record:

```
record Address(String city, String zip) {
}

record Employee(String name, Address address) {
}

class NestedRecordExampleUsage {
    public static void main(String[] args) {
        Employee e1 = new Employee("Ada", new Address("Istanbul", "34000"));
        Employee e2 = new Employee("Ada", new Address("Istanbul", "34000"));

        System.out.println(e1);              // Employee[name=Ada, address=Address
[city=Istanbul, zip=34000]]
        System.out.println(e1.equals(e2));    // true — the nested Address.equals() is used
too
        System.out.println(e1.address().city()); // Istanbul
    }
}
```

The `Employee` record's `address()` accessor returns a value of type `Address`; this means the `equals()` / `hashCode()` / `toString()` chain naturally works **recursively** — as long as `Address`'s own `equals()` is correct (which is automatic for a record), `Employee.equals()` also works correctly, because comparing that component calls `Address.equals()`.

Serialization and Reflection

Unlike enums, a record is **not automatically** `Serializable` — you need to declare this explicitly, just like an ordinary class. When you do, an important record-specific difference emerges: unlike classic Java serialization, deserialization doesn't populate fields directly via reflection — it **calls the canonical constructor**:

```
import java.io.*;

record Score(String player, int points) implements Serializable {

    // The validation in the compact constructor also runs during deserialization --
    // unlike a classic class, where a custom readObject() could bypass it.
    Score {
        if (points < 0) {
            throw new IllegalArgumentException("points negatif olamaz: " + points);
        }
    }
}

class SerializableRecordExampleUsage {
    public static void main(String[] args) throws Exception {
        Score original = new Score("Ada", 100);

        ByteArrayOutputStream bytes = new ByteArrayOutputStream();
        try (ObjectOutputStream out = new ObjectOutputStream(bytes)) {
            out.writeObject(original);
        }

        Score restored;
        try (ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(bytes.
toByteArray()))) {
            restored = (Score) in.readObject();
        }

        System.out.println(restored);                  // Score[player=Ada, points=100]
        System.out.println(original.equals(restored));  // true
    }
}
```

> This provides an important practical security benefit: the compact constructor validation from "Constructors" (e.g. the `points < 0` check) **cannot be bypassed** during deserialization. In a classic `Serializable` class, unless you write a custom `readObject()`, an attacker with a controlled byte stream could "construct" an invalid object by bypassing validation — for a record, this path is closed from the start.

Reflection also offers two new record-specific tools: `Class.isRecord()` and `Class.getRecordComponents()` — the latter lets you list component names and types at runtime (JSON serializers, ORMs, and validation libraries use exactly this to recognize records):

```
import java.lang.reflect.RecordComponent;

record Score2(String player, int points) {
}

class ReflectionExample {
    public static void main(String[] args) {
        Class<Score2> type = Score2.class;

        System.out.println(type.isRecord()); // true

        for (RecordComponent component : type.getRecordComponents()) {
            System.out.println(component.getName() + " : " + component.getType().
getSimpleName());
        }
        // player : String
        // points : int
    }
}
```

Expected Output

```
true
player : String
points : int
```

> `getRecordComponents()` returns `null`, not an empty array, if the class is **not** a record — always check with `isRecord()` before calling it, or you may get an unexpected `NullPointerException`.

Best Practices

Let's turn everything we've seen so far into concrete guidance on when to use a record and when not to.

Use a record for:

- DTOs, request/response models (see "Real-World Examples")
- Value objects — monetary amounts, coordinates, ranges
- Data models you'll pattern-match with `switch` / `instanceof` — see the **Record Patterns** appendix
- When a method needs to return more than one value (as a dedicated "result" type)

Don't use a record for:

- JPA/Hibernate entities (we covered why in "Record vs Class")
- Objects whose internal state needs to change over time (e.g. a builder itself, or a cache entry holding a counter)

- Structures with a large number of components (more than 6–7) — this is usually a signal to "group related fields into a nested record" (see "Nested Records")

Design recommendations:

- Always make a defensive copy in the compact constructor for mutable components (`List`, `Map`, `Set`) (see "Immutability")
- Keep complex validation logic in the compact constructor, don't leave it to the caller's responsibility
- Avoid array components (see "Immutability")
- You don't need a `Record` suffix in naming — in this lesson we used it (`PersonClassic` / `PersonRecord`) to distinguish multiple variants of `Point`; in real code, plain names like `Person`, `Point`, `Range` are preferred

> If your project uses Lombok, you may need to choose between `@Data` / `@Value` and a record — the short answer: use a **record** for newly written data types that genuinely need to be immutable and that you want compiled without needing an external annotation processor; use **Lombok** for classes that need to stay mutable, like JPA entities, or that need features a record doesn't support (like `@Builder`). We cover the detailed comparison in the "Record vs Lombok" appendix.

Common Mistakes

Let's gather the pitfalls we ran into one by one along the way, plus a couple of new ones.

1. Using an array as a component. `equals()` uses reference equality for array components, not `Arrays.equals()` — we covered this in detail in "Immutability".

2. Holding onto a mutable object as-is. Taking an `ArrayList` / `HashMap` reference still held by the calling code without copying it in the compact constructor leaves a visible "immutability" guarantee with a real hole in it.

3. Trying to make a record a JPA entity. The requirements of a no-args constructor and mutable fields conflict directly with a record's nature.

4. Expecting `getX()` / `getY()`. Record accessors use the component's name, not the Java Bean prefix — we covered this in "Creating Your First Record".

5. Assuming different record types with the "same shape" are equal. `equals()` first checks whether the runtime class is **exactly the same** — even if two records' components look identical, they're never equal if their types differ:

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

Point p = new Point(1, 2);
Coordinate c = new Coordinate(1, 2);
System.out.println(p.equals(c)); // false — a Coordinate is not a Point
```

6. Designing a record with "we'll extend it later" in mind. Records are implicitly `final` and cannot be extended (see "Record vs Class") — if you want to share behavior, composition (making one record a component of another, see "Nested Records") or implementing a common interface (see "Implementing Interfaces") is the right path.

Real-World Examples

The most natural habitat for records is a Spring Boot application's **boundary layer** (controllers) and **read models**. A user-creation request and response are typically modeled like this:

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

// As we saw in the "Components" section, validation annotations are automatically
// applied to the field, the constructor parameter, and the accessor alike.
record CreateUserRequest(
        @NotBlank String fullName,
        @NotBlank @Email String email
) {
}
```

```
import java.time.Instant;

record UserResponse(Long id, String fullName, String email, Instant createdAt) {

    // Static factory: a common pattern for keeping the entity -> response
    // conversion in one place (the same pattern we saw in "Static Members").
    static UserResponse from(Long id, String fullName, String email, Instant createdAt) {
        return new UserResponse(id, fullName, email, createdAt);
    }
}
```

Using them in a controller is identical to an ordinary class — Spring deserializes records for `@RequestBody` just like a normal class (via Jackson):

```
import jakarta.validation.Valid;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.time.Instant;

@RestController
class UserController {

    // In a real application a UserService would be injected here; to keep the
    // example simple, we build the UserResponse directly.
    @PostMapping("/api/users")
    ResponseEntity<UserResponse> create(@Valid @RequestBody CreateUserRequest request) {
        UserResponse response = UserResponse.from(1L, request.fullName(), request.email(),
Instant.now());
        return ResponseEntity.ok(response);
    }
}
```

> For the `@NotBlank` / `@Email` constraints on components to be triggered by `@Valid`, the `spring-boot-starter-validation` dependency must be on the classpath — `spring-boot-starter-web` doesn't bring it in automatically.

> Jackson (Spring Boot's default JSON library) automatically recognizes a record's **single** constructor — the canonical constructor — and maps JSON fields directly to it; you generally don't need to add `@JsonCreator` or `@JsonProperty`. This makes using records as request/response DTOs less code and less risk of "forgetting something" compared to a hand-written POJO.

> Spring Data JPA also accepts records as the target type in JPQL constructor expressions written with `@Query` (`SELECT new com.cdurgun.learning.dto.UserSummary(u.id, u.email) FROM User u`) and in interface-based projections — even though we can't make the entity itself a record, we can comfortably model a **read-only view/projection** derived from it with a record.

Interview Questions

What is a record, and how is it fundamentally different from a regular class? A record is a special kind of class designed to carry immutable data; the compiler automatically generates the constructor, accessors, `equals()`, `hashCode()`, and `toString()`. It's implicitly `final`, extends `java.lang.Record`, and all its fields are `private final`.

Why can't a record be used as a JPA entity? JPA requires a no-args constructor and a non-`final`, mutable class to create proxies; a record's design (immutable, single constructor, implicitly `final`) directly conflicts with these requirements.

What is a compact constructor, and when is it used? It's a special form of the canonical constructor that lets you write only validation/normalization logic without repeating the parameter list or assignments; the assignments are done implicitly by the compiler after the block. It's typically used for input validation and defensive copying (`List.copyOf()`).

Are records `Serializable`? No, unlike enums this isn't automatic — you need to explicitly write `implements Serializable`. When you do, deserialization, unlike ordinary classes, calls the canonical constructor, which guarantees that the compact constructor's validation also runs during deserialization.

Can a record extend another class? No — every record implicitly extends `java.lang.Record`, and since Java supports single inheritance, it can't extend another class. It can, however, implement as many interfaces as it wants.

Can you add an extra instance field to a record's body? No, this is a compile error — a record's state consists entirely of its component list. Static fields aren't subject to this restriction.

What is the relationship between records and sealed interfaces? Defining all the permitted subtypes of a `sealed interface` as records lets the compiler guarantee, in a modern `switch` pattern match, that all possible cases are covered — we go into detail on this in the **Record Patterns** appendix.

Summary and Cheat Sheet

A record, made permanent in Java 16, is a special kind of class for immutable data carriers, defined in a single line, that delegates the constructor, accessors, `equals()`, `hashCode()`, and `toString()` to the compiler. Key points:

- Component list = field + accessor + canonical constructor parameter, all in one
- Implicitly `final`, extends `java.lang.Record`, all fields `private final`
- `equals()` / `hashCode()` / `toString()` are auto-generated based on components (`Float.compare()` / `Double.compare()` semantics for `float` / `double`)
- Immutability is **shallow** — make a defensive copy in the compact constructor for mutable components, avoid array components
- Compact constructor for validation/normalization; extra constructors must delegate to the canonical one
- Static fields/methods are free, extra **instance** fields are forbidden
- Can implement interfaces, cannot extend
- Can be nested (implicitly `static`)

- Not `Serializable` automatically; if declared, deserialization calls the canonical constructor
- Ideal for: DTOs, request/response, value objects, pattern-matching targets
- Avoid for: JPA entities, classes needing mutable state

Quick reference:

```
// Basic definition
record Point(int x, int y) {}

// Compact constructor (validation/normalization)
record Point(int x, int y) {
    Point {
        if (x < 0 || y < 0) throw new IllegalArgumentException("cannot be negative");
    }
}

// Static factory + constant
record Point(int x, int y) {
    static final Point ORIGIN = new Point(0, 0);
    static Point of(int x, int y) { return new Point(x, y); }
}

// Implementing an interface
record Point(int x, int y) implements Comparable<Point> {
    public int compareTo(Point o) { return Integer.compare(x, o.x); }
}

// Generic record
record Pair<A, B>(A first, B second) {}

// Nested record
record Address(String city, String zip) {}
record Employee(String name, Address address) {}
```

Appendix: Record vs Lombok

This project already has Lombok among its dependencies — so let's give a concrete answer to "why did we write a record instead of `@Data` or `@Value`?"

Lombok is an **annotation processor** that runs at compile time: without changing your source code, it injects members like the constructor/getter/setter/`equals()` into the `.class` file. `@Value` is the closest to a record in intent — it produces an immutable class:

```
// With Lombok: @Value produces an immutable class
import lombok.Value;

@Value
public class PersonLombok {
    String name;
    int age;
}

// With a record: same result, at the language level
record PersonRecord(String name, int age) {
}
```

Both end up generating a constructor, accessors, `equals()`, `hashCode()`, `toString()` — but the underlying mechanism and flexibility diverge in important ways:

- **Mechanism:** A record is part of `javac` itself — no extra dependency or IDE plugin is needed. Lombok is an external library; the IDE needs the Lombok plugin installed to display the code correctly (we assume it's already installed in this project, but it's an extra setup step for a new contributor).
- **Accessor name:** A record produces `name()`; Lombok's `@Value`/`@Data` follows the Java Bean convention and produces `getName()`. As we noted in "Creating Your First Record", this is a deliberate design difference — Lombok prioritizes compatibility with older Bean-based frameworks (some reflection-based serializers, form-binding libraries).
- **Mutability choice:** A record is always immutable. With Lombok, this is a choice — `@Value` is immutable, `@Data` produces a mutable class (getters **and** setters). So Lombok is a single tool for both immutable and mutable data classes.
- **Inheritance:** A record is implicitly `final` and can't be extended. Lombok's generated class is an ordinary class — you can extend it, add extra fields/methods (at the cost of breaking the immutability guarantee).
- **Builder:** Lombok's `@Builder` gives you a ready-made builder API for multi-component objects. A record has no built-in builder — you'd need to write one by hand or add a separate annotation processor (e.g. an external "record builder" library).
- **Pattern matching:** Only genuine records integrate directly with Java's `switch`/`instanceof` pattern matching (see the **Record Patterns** appendix) — classes generated by Lombok can't take advantage of this, because the compiler doesn't recognize them as records.
- **Validation guarantee:** As we saw in "Constructors", extra constructors in a record must delegate to the canonical one — the single entry point is enforced by the compiler. With Lombok, a similar guarantee requires you to hand-write the constructor and apply `@Value` to the fields, which relies on discipline rather than compiler enforcement.

> As a rule of thumb: use a **record** for newly written data types that genuinely need to be immutable and may be a target of pattern matching; use **Lombok** for complex objects that need a builder API, classes that need mutable state, or classes added to an older codebase that already depends on Lombok and needs the Bean convention. The two aren't mutually exclusive — they can coexist in the same project, just as they do here.

Appendix: Record Patterns (Java 21)

We already used the modern `switch` syntax in the Enum topic; Java 21 takes this syntax a step further for records: **record patterns** let you both type-check and deconstruct a record into its components in a single line.

In its simplest form, with `instanceof`:

```
Object obj = new Point(3, 4);

if (obj instanceof Point(int x, int y)) {
    System.out.println("x=" + x + ", y=" + y); // x and y are directly usable here
}
```

In the classic approach, you'd first do an `instanceof Point` type check and then access the components with `((Point) obj).x()` — a record pattern merges these two steps into one line, and exposes `x` / `y` directly as usable local variables in the result.

It shows its real power when you model all the subtypes of a `sealed interface` as records and combine it with `switch`:

```
sealed interface Shape permits Circle, Rectangle, Square {
}

record Circle(double radius) implements Shape {
}

record Rectangle(double width, double height) implements Shape {
}

record Square(double side) implements Shape {
}

class SealedShapeExample {

    static double area(Shape shape) {
        return switch (shape) {
            case Circle(double r) -> Math.PI * r * r;
            case Rectangle(double w, double h) -> w * h;
            case Square(double s) -> s * s;
        };
    }

    public static void main(String[] args) {
        System.out.printf("%.2f%n", area(new Circle(2)));      // 12.57
        System.out.println(area(new Rectangle(3, 4)));          // 12.0
        System.out.println(area(new Square(5)));                // 25.0
    }
}
```

> There's **no** `default` branch in the switch expression above — because `Shape` is `sealed` and has only three `permits`-ed subtypes, the compiler can verify that all possible cases are covered (just like an exhaustive switch over an enum, see the "Usage with switch" section in the Enum topic). If a new subtype is added to `Shape` and this switch isn't updated, the build **fails to compile** — a gap caught at compile time, not at runtime.

Record patterns can also be used **nested** — let's rewrite the `Employee`/`Address` example from "Nested Records" to access the components in a single line:

```
record Address(String city, String zip) {
}

record Employee(String name, Address address) {
}

class NestedPatternExample {

    static String describe(Object obj) {
        if (obj instanceof Employee(String name, Address(String city, String zip))) {
            return name + " - " + city + " (" + zip + ")";
        }
        return "unknown";
    }

    public static void main(String[] args) {
        Employee e = new Employee("Ada", new Address("Istanbul", "34000"));
        System.out.println(describe(e)); // Ada - Istanbul (34000)
    }
}
```

Expected Output

```
Ada - Istanbul (34000)
```

Finally, when you want to add an extra condition to a pattern, you use a **guarded pattern** (the `when` keyword):

```
static String describe(Shape shape) {
    return switch (shape) {
        case Circle(var r) when r > 100 -> "A huge circle";
        case Circle(var r) -> "A circle with radius " + r;
        case Rectangle(var w, var h) -> "A rectangle (" + w + "x" + h + ")";
        case Square(var s) -> "A square with side " + s;
    };
}
```

> The `case Circle(var r) when r > 100 -> ...` line must come **before** the general `case Circle(var r) -> ...` line — switch branches are tried top to bottom, just like a classic `if/else if` chain. If the order were reversed, the general `Circle` branch would always match first, and the guarded (`when`) branch would never run.