

SQL Types vs Java Types

- for typed getters (`getInt`, `getString`, `getBoolean`, ...), the driver converts the column's value to that Java type
 - primitive getters (`getInt`, `getDouble`, `getBoolean`) return 0 or false for NULL
 - NULL trap – 0/false are legal values
 - getters that return objects (`getString`, `getBigDecimal`) return null
- for `getObject(col, Type.class)`, name the Java type you want
 - returns null for NULL
 - throws `SQLException` if the driver can't do the conversion
- avoid `getDate` – use `getObject` for `java.time` types
 - modern time API uses `LocalDate`, `ZonedDateTime`, `Instant`
- use `getObject` for nullable numbers (e.g. `getObject(col, Integer.class)`)
- avoid plain `getObject(col)` – the driver picks the type and you cast

SQL Types vs Java Types

column	SQL type	Java type	getter
book_id	INT	int / Integer	getInt
title	VARCHAR	String	getString
published_on	DATE NULL	LocalDate	getObject(col, LocalDate.class)
replacement_cost	DECIMAL(6,2) NULL	BigDecimal	getBigDecimal
available	BOOLEAN	boolean	getBoolean

```
int id = rs.getInt("book_id"); // NOT NULL column: a primitive is safe
String title = rs.getString("title"); // null for SQL NULL
BigDecimal cost = rs.getBigDecimal("replacement_cost"); // null for SQL NULL
// no getLocalDate, so name the type
LocalDate pub = rs.getObject("published_on", LocalDate.class);
```

LocalDate

- holds a year, month, day (no time or time zone)
- create with `LocalDate(2026, 10, 12)`, `LocalDate.parse("2026-10-12")`, or `LocalDate.now()`
- immutable, so operations like `due.plusDays(7)` returns the new date – `due` is not changed
- compare with `isBefore`, `isAfter`, `equals`
- also `LocalTime`, `LocalDateTime`

Getting the New ID

- a create endpoint such as `POST /books` responds with 201 Created and the new ID
 - `Statement.RETURN_GENERATED_KEYS` means that generated keys (`AUTO_INCREMENT`) are handed back in a `ResultSet` instead of needing a second query to retrieve

```
PreparedStatement ps = conn.prepareStatement(
    "INSERT INTO BOOKS (isbn, title, author) VALUES (?, ?, ?)",
    Statement.RETURN_GENERATED_KEYS);
ps.setString(1, "9780345391803");
ps.setString(2, "The Hitchhiker's Guide to the Galaxy");
ps.setString(3, "Douglas Adams");
ps.executeUpdate();
ResultSet keys = ps.getGeneratedKeys();
keys.next();
int bookId = keys.getInt(1); // the AUTO_INCREMENT value
```

Resource Cleanup

- leaks exhaust MySQL's connection limit and the application stops works
- use a try-with-resources to ensure everything gets closed
 - things that hold resources – connections, statements, and result sets – are declared in a `()` section
 - automatically closed in reverse order, even on an exception

```
String sql = "SELECT book_id, title, author, available FROM BOOKS "+
            "ORDER BY title";
try (Connection conn = Db.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql);
    ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
        int id = rs.getInt("book_id");
        String title = rs.getString("title");
        String author = rs.getString("author");
        boolean available = rs.getBoolean("available");
        System.out.println(id + ": " + title + " by " + author
            + (available ? "" : " (checked out)");
    }
} // rs, ps, conn are closed (in that order) even if an exception is thrown
```

18

When Something Goes Wrong

- every JDBC call can throw `SQLException`
- for now, just propagate – we'll handle errors later

CPSC 343: Database Theory and Practice • Fall 2026

39