Java 11–21:

This guide explains the most important Java features (from Java 11 to Java 21) that help you write less code.

1️⃣ Switch Expressions (Java 14+)

What is it?

Switch expressions allow you to use `switch` as an expression that directly returns a value, with a concise syntax and no need for `break` statements.

Before Java 14

```
String dayType;
switch (day) {
    case "MONDAY":
    case "TUESDAY":
    case "WEDNESDAY":
    case "THURSDAY":
    case "FRIDAY":
        dayType = "Weekday";
        break;
    case "SATURDAY":
    case "SUNDAY":
        dayType = "Weekend";
        break;
    default:
        dayType = "Unknown";
}
```

Explanation:
You have to declare a variable outside the switch, assign it inside each case, and remember to use `break` to avoid fall-through bugs.

After Java 14

```
String dayType = switch (day) {
    case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" ->
"Weekday";
    case "SATURDAY", "SUNDAY" -> "Weekend";
    default -> "Unknown";
};
```

Explanation:
With switch expressions, you write less code. You can return values directly from cases using `->`, and group cases by commas. No `break` needed.

With Logic (using `yield`)

```java
int score = 85;
String grade = switch (score / 10) {
    case 10, 9 -> "A";
    case 8 -> "B";
    case 7 -> "C";
    case 6 -> "D";
    default -> {
        System.out.println("Score is below passing.");
        yield "F";
    }
};
```

Explanation:
If your case needs more code (like logging), you can use a code block and `yield` to return a value.

Benefits:

- No need for break statements.
- Directly returns values, reducing boilerplate.
- Safer (no accidental fall-through).
- Easier to read and maintain.

2️⃣ Text Blocks (Java 15+)

What is it?

Text blocks use triple quotes (`"""`) to easily write multi-line string literals, with preserved formatting and indentation.

Before Java 15

```java
String html = "<html>\n" +
              "  <body>\n" +
              "    <h1>Hello, World!</h1>\n" +
              "  </body>\n" +
              "</html>";
```

Explanation:
Multi-line strings required awkward concatenation and escaped newlines, making them hard to read and maintain.

After Java 15

```
String html = """
    <html>
      <body>
        <h1>Hello, World!</h1>
      </body>
    </html>
    """;
```

Explanation:
Text blocks make multi-line strings readable and maintainable. No more `\n` or `+`—just write the string as it should appear.

Benefits:

- Easier to write and read multi-line strings.
- Great for JSON, HTML, SQL, configs.
- No need for manual escaping or concatenation.

3️⃣ Records (Java 16+)

What is it?

Records are a concise way to declare immutable data classes.
They auto-generate constructors, accessors, `equals`, `hashCode`, and `toString`.

Before Java 16

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    @Override public boolean equals(Object o) { /* ... */ }
    @Override public int hashCode() { /* ... */ }
    @Override public String toString() { /* ... */ }
}
```

Explanation:
To make a simple data holder, you have to write lots of boilerplate for fields, constructors, getters, and utility methods.

After Java 16

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

Explanation:
A record declaration automatically creates all the code you need for an immutable data class.
You get final fields, a constructor, getters (`name()`, `age()`), and meaningful `equals`, `hashCode`, and `toString`.

Usage:

```
Person p = new Person("Yasmine", 25);
System.out.println(p.name()); // "Yasmine"
System.out.println(p);        // Person[name=Yasmine, age=25]
```

Benefits:

- No boilerplate for simple data holders.
- Immutability by default.
- Cleaner and safer code.

4️⃣ Pattern Matching

4.1 Pattern Matching for `instanceof` (Java 16+)

What is it?

Pattern matching lets you type-check and cast in a single step, making code shorter and safer.

Before Java 16

```
Object obj = "hello";
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toUpperCase());
}
```

Explanation:
You have to check the type with `instanceof`, then cast manually, which is repetitive and error-prone.

After Java 16

```
Object obj = "hello";
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

Explanation:
With pattern matching, you declare the variable inside the condition. The type is checked and casted automatically.

Benefits:

- Fewer lines.
- No manual casting.
- Safer and easier to read.

4.2 Pattern Matching for `switch` (Java 17+, 21)

What is it?

You can use `switch` to match not only values, but types and structure (like records), and add conditions.

Before Java 17

```
Object o = "Java";
if (o instanceof String) {
    System.out.println("String: " + ((String) o).length());
} else if (o instanceof Integer) {
    System.out.println("Integer: " + o);
} else {
    System.out.println("Other");
}
```

Explanation:
Type-specific logic requires multiple `if-else` blocks and manual casting.

After Java 17+

```
Object o = "Java";
String result = switch (o) {
    case String s -> "String of length " + s.length();
    case Integer i -> "Integer: " + i;
    case null -> "Null!";
    default -> "Other";
};
System.out.println(result); // String of length 4
```

Explanation:
Switch can match on type and bind a variable, making logic concise and readable.

Matching records (Java 21)

```
record Point(int x, int y) {}
Object obj = new Point(4, 4);
String desc = switch (obj) {
    case Point(int x, int y) when x == y -> "Diagonal";
    case Point(int x, int y) -> "Point (" + x + "," + y + ")";
    default -> "Unknown";
};
System.out.println(desc); // Diagonal
```

Explanation:
Switch can deconstruct records and add conditions with `when`, so you can match structure and values in one expression.

Benefits:

- Concise type and structure checks.
- No manual casts or long `if-else` chains.
- Expressive, safe, and maintainable logic.

Summary Table

Feature	Before Example	After Example	Explanation & Benefit
Switch Expressions	Verbose with breaks/variables	Direct value with `->`	Safer, shorter, no fall-through
Text Blocks	`\n` and `+` for multi-line	`""" ... """`	Easy, readable multi-line strings

Feature	Before Example	After Example	Explanation & Benefit
Records	Full class with boilerplate	`record Name(Type...)`	All code auto-generated, less typing
Pattern Matching	Manual instanceof & cast	`if (x instanceof Type t)`	Fewer lines, safer, clearer
Switch Pattern Match	Multiple if-else, manual cast	`switch (x) { case Type t -> ... }`	Clean, expressive, multi-type logic