switch

The second topic in the Control Flow category: classic `case` / `break` syntax and the fall-through trap, the modern arrow (`->`) syntax, switch EXPRESSIONS and `yield`, multiple labels with enum exhaustiveness checking, and using `switch` with `String`/enum.

Beginner 20 min

Last updated: October 19, 2026
Read online: https://www.learnforgex.com/en/topics/switch

---

switch

The second topic in the Control Flow category: switch -- a control structure that compares a single value against several possible constants and branches accordingly. A long "else if Chains" (see the "if / else" lesson) can always be rewritten as a `switch`, but for this specific case -- comparing ONE variable against MULTIPLE constant values -- `switch` offers syntax that's both shorter and lets the compiler check more (for example, confirming that every constant of an enum has been handled). With Java 21 as the target, this lesson covers both the classic `switch` syntax and the modern arrow (`->`) syntax that arrived as a preview in 2017 and became permanent in Java 14 (2020), along with the `switch` EXPRESSION.

What Is switch?

`switch` compares a value in parentheses (an `int`, a `String`, an enum constant, etc.) against a series of `case` labels in order -- the matching `case` runs, and if none match, the (optional) `default` runs. Java offers two different syntaxes: the CLASSIC syntax (`case value:` with an optional `break;`) and the modern arrow syntax (`case value -> expression;`, since Java 14). `switch` can also be used as an EXPRESSION -- evaluating directly to a value that can be assigned to a variable (see "Switch Expressions and yield").

Why Does It Exist?

An `else if` chain (see "else if Chains") that repeatedly compares the same variable against many constants technically always works, but has two problems: (1) the variable name gets typed OVER AND OVER in every condition (`if (day == 1) ... else if (day == 2) ...`), hurting readability; (2) the compiler has NO WAY of knowing this is the "compare one value

against a set of constants" pattern, so it can't check things like whether ALL constants of an enum are covered. `switch` bakes this pattern directly into the language, giving both shorter syntax and compiler-assisted exhaustiveness checking.

History

`switch` carried over from C (1972) to Java (1995) almost verbatim -- the classic `case`/`break` syntax and its fall-through behavior (see "Fall-Through: The Cost of Forgetting break") are a direct inheritance from C. That behavior remained a real source of bugs for decades; Java introduced the modern arrow syntax (`->`, NO fall-through) as a PREVIEW feature in 2017 via JEP 325, which became a permanent language feature in Java 14 (2020) -- the same JEP also added the ability to use `switch` as an EXPRESSION, along with the `yield` keyword.

Classic switch Syntax

In the classic syntax, each `case` is a VALUE label followed by the statements to run; `break` exits the `switch`. `default` runs when no `case` matches -- it isn't required, but it's good practice (see "Common Mistakes").

```
public class SwitchBasicsExample {
    public static void main(String[] args) {
        int day = 3;
        String name;

        switch (day) {
            case 1:
                name = "Monday";
                break;
            case 2:
                name = "Tuesday";
                break;
            case 3:
                name = "Wednesday";
                break;
            default:
                name = "Unknown";
                break;
        }

        System.out.println("Day " + day + " -> " + name);
    }
}
```

Expected Output

```
Day 3 -> Wednesday
```

Fall-Through: The Cost of Forgetting break

In the classic syntax, if a `case` is NOT followed by `break`, execution "falls through" into the NEXT case as well -- regardless of whether it matches -- until a `break` or the end of the `switch` is reached. In rare cases this is used deliberately (when several cases should share the same behavior), but most of the time a missing `break` is a real bug.

```
public class FallThroughExample {
    public static void main(String[] args) {
        System.out.println("Without break (BUGGY):");
        printDayBuggy(2);

        System.out.println();
        System.out.println("With break (FIXED):");
        printDayFixed(2);
    }

    // Missing break statements -- once a case matches, execution keeps
    // running into every case BELOW it until a break (or the end) is hit.
    private static void printDayBuggy(int day) {
        switch (day) {
            case 1:
                System.out.println("Monday");
            case 2:
                System.out.println("Tuesday");
            case 3:
                System.out.println("Wednesday");
            default:
                System.out.println("Unknown");
        }
    }

    private static void printDayFixed(int day) {
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            default:
                System.out.println("Unknown");
                break;
        }
    }
}
```

Expected Output

```
Without break (BUGGY):
Tuesday
Wednesday
Unknown

With break (FIXED):
Tuesday
```

> If you use fall-through DELIBERATELY (e.g. `case 6: case 7: // weekend`), leaving a comment explaining it is a strongly recommended practice -- a reader has no way to tell a deliberate design from a forgotten `break`. The modern arrow syntax (see "The Modern Arrow Syntax") removes this ambiguity entirely.

The Modern Arrow Syntax

The `->` syntax, permanent since Java 14, runs EACH case as its own independent branch -- there is NO fall-through, and `break` is NOT needed. Multiple values can be attached to a single branch, separated by commas (`case 1, 2, 3 -> ...`).

```
public class ArrowSwitchExample {
    public static void main(String[] args) {
        printDayType(3);
        printDayType(6);
        printDayType(7);
    }

    // The arrow (->) form never falls through -- each case runs ONLY its own
    // branch, no break needed. Multiple labels can share one branch.
    private static void printDayType(int day) {
        switch (day) {
            case 1, 2, 3, 4, 5 -> System.out.println(day + " is a weekday.");
            case 6, 7 -> System.out.println(day + " is a weekend day.");
            default -> System.out.println(day + " is not a valid day.");
        }
    }
}
```

Expected Output

```
3 is a weekday.
6 is a weekend day.
7 is a weekend day.
```

Switch Expressions and yield

`switch` can also be written as an EXPRESSION -- evaluating directly to a value that can be assigned to a variable (`String result = switch (x) { ... };`), similar to the ternary

operator (see "The Ternary Operator" in the "if / else" lesson) but supporting more branches. With arrow syntax, a single expression directly produces the value; when a branch needs a BLOCK body with several steps, the produced value must be specified with the `yield` keyword.

```
public class SwitchExpressionExample {
    public static void main(String[] args) {
        System.out.println("1 -> " + classify(1));
        System.out.println("3 -> " + classify(3));
        System.out.println("6 -> " + classify(6));
        System.out.println("9 -> " + classify(9));
    }

    // A switch EXPRESSION evaluates directly to a value -- no need for a
    // separate variable that gets assigned inside each branch.
    private static String classify(int day) {
        return switch (day) {
            case 1, 2, 3, 4, 5 -> "weekday";
            case 6, 7 -> "weekend";
            default -> {
                // A block body needs `yield` to produce the expression's value.
                System.out.println("  (unexpected value logged: " + day + ")");
                yield "invalid";
            }
        };
    }
}
```

Expected Output

```
1 -> weekday
3 -> weekday
6 -> weekend
  (unexpected value logged: 9)
9 -> invalid
```

Multiple Labels and Exhaustiveness

Attaching multiple values to one `case` with commas (`case SATURDAY, SUNDAY -> ...`) reduces duplication. A switch EXPRESSION over an enum compiles WITHOUT a `default` as long as every constant of the enum is covered by at least one `case` -- the compiler verifies this with EXHAUSTIVENESS checking. This means that if a new constant is added to the enum later and this `switch` is forgotten, you get a COMPILE ERROR instead of a silent runtime bug.

```
public class MultipleLabelsAndDefaultExample {

    enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        System.out.println(describe(Season.WINTER));
        System.out.println(describe(Season.SPRING));
        System.out.println(describe(Season.SUMMER));
        System.out.println(describe(Season.FALL));
    }

    // A switch EXPRESSION over an enum can skip `default` entirely as long as
    // EVERY constant is covered -- the compiler checks this exhaustiveness for
    // you. Adding a new enum constant later without updating this switch
    // becomes a COMPILE ERROR, not a silent bug.
    private static String describe(Season season) {
        return switch (season) {
            case WINTER, FALL -> "cold";
            case SPRING, SUMMER -> "warm";
        };
    }
}
```

Expected Output

```
cold
warm
warm
cold
```

> This exhaustiveness check does NOT apply when `switch` is used as a statement (classic or arrow syntax, but not producing a value) -- the compiler won't enforce it. If you want the exhaustiveness guarantee, writing `switch` as a value-producing expression (not just for its side effects) should be a deliberate choice.

Using switch with String and Enum

Besides primitives like `int` / `char`, `switch` also works with `String` and enum constants. A `switch` on a `String` compares CONTENT (like `.equals()`, NOT like `==`) -- see "Comparison Operators" in the "if / else" lesson. In a `switch` over an enum, `case` labels are conventionally written WITHOUT the enum name prefix (`case ADMIN ->` rather than `case Role.ADMIN ->`) -- the same form the enum lesson's own "Usage with switch" section uses. Before Java 21 the prefixed form was a COMPILE ERROR; since Java 21 it is accepted too, but the short form remains the usual style.

```
public class SwitchOnStringAndEnumExample {

    enum Role { ADMIN, EDITOR, VIEWER }

    public static void main(String[] args) {
        System.out.println(permissionsFor("admin"));
        System.out.println(permissionsFor("editor"));
        System.out.println(permissionsFor("guest"));

        System.out.println(roleLabel(Role.ADMIN));
        System.out.println(roleLabel(Role.VIEWER));
    }

    // Switching on a String compares by CONTENT (like .equals(), not ==) --
    // no need to worry about the == vs equals() trap from the "Comparison
    // Operators" section of the if / else lesson.
    private static String permissionsFor(String role) {
        return switch (role) {
            case "admin" -> "read, write, delete";
            case "editor" -> "read, write";
            default -> "read only";
        };
    }

    private static String roleLabel(Role role) {
        return switch (role) {
            case ADMIN -> "Administrator";
            case EDITOR -> "Editor";
            case VIEWER -> "Viewer";
        };
    }
}
```

Expected Output

```
read, write, delete
read, write
read only
Administrator
Viewer
```

Best Practices

- **Never forget `break` at the end of each `case` in classic syntax** -- or just prefer the fall-through-free modern arrow syntax entirely.
- **If you use fall-through deliberately, say so explicitly with a comment** -- otherwise it reads as a forgotten `break`.
- **When writing a `switch` over an enum that covers every constant, skip `default` and let exhaustiveness checking work for you** -- getting a compile error when a new constant is added is far better than a silent bug.
- **Write value-producing switches as an EXPRESSION, rather than the classic statement-plus-temporary-variable pattern.**

- **If a long `else if` chain compares ONE variable against multiple constants, consider converting it to a `switch`.**

Common Mistakes

- **Forgetting `break` at the end of a `case` in classic syntax, causing unwanted fall-through.** Execution "falls" into the next cases as well, even if they don't match.
- **Assuming a `switch` on a `String` compares like `==`.** It actually compares content -- the `String` reference trap does NOT apply here.
- **Prefixing enum `case` labels with the enum name on a Java version older than 21** (`case Role.ADMIN ->` instead of `case ADMIN ->`). There it is a compile error; since Java 21 both forms compile, and the unprefixed one is the conventional choice.
- **Forgetting `yield` in a switch EXPRESSION's block body.** The compiler has no way to know which value the block should produce.

Summary, Cheat Sheet, and Glossary

`switch` is a control structure that compares a single value against multiple constants -- a shorter, compiler-assisted alternative to a long `else if` chain. Classic syntax uses `case` / `break`, and forgetting `break` causes FALL-THROUGH; the modern arrow (`->`) syntax has no fall-through. `switch` can be written as an EXPRESSION that evaluates to a value; block bodies need `yield`. A switch EXPRESSION covering every constant of an enum can compile without `default` -- the compiler checks exhaustiveness.

Quick reference:

```
// Classic
switch (day) {
    case 1: result = "Monday"; break;
    default: result = "Unknown"; break;
}

// Modern arrow syntax
switch (day) {
    case 1, 2, 3, 4, 5 -> System.out.println("Weekday");
    case 6, 7 -> System.out.println("Weekend");
}

// Switch expression + yield
String type = switch (day) {
    case 1, 2, 3, 4, 5 -> "weekday";
    case 6, 7 -> "weekend";
    default -> {
        yield "invalid";
    }
};
```

Glossary

Fall-Through — In classic `switch` syntax, execution continuing into the next case(s) until a `break` is reached.

Switch Expression — A form of `switch` that evaluates directly to a value, which can be assigned to a variable.

yield — The keyword that specifies the value to produce from a switch expression's BLOCK body.

Exhaustiveness — The compiler verifying that a switch expression covers every possible value it operates on (e.g. every constant of an enum) without needing `default`.

As a more advanced topic, matching `switch` against object TYPES and a record's components (pattern matching) is covered separately in "Appendix: Record Patterns (Java 21)" in the "Record" lesson.