Chapter 3.1: Control Structures

Object Oriented Programming with Java

Sequential execution | if-else | else-if chains | Nested decisions | switch-case

The for loop | while and do-while | Nested loops | break and continue

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_03_01_Control_Structures.html

INTRODUCTION

Learning Outcomes

Flow of control

Explain how sequence, selection and loops direct the flow of a program.

Decisions

Write conditional code with `if`, `if-else`, `else if` chains, nested decisions and `&&` / `||` conditions.

switch-case

Compare a value against constants; explain what `break` does and how “fall-through” works.

Loops

Use `for`, `while` and `do-while` where each fits; avoid infinite loops and off-by-one errors.

Nested loops

Count how many times nested loops run in patterns, tables and digit examples.

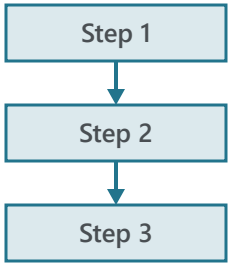
Tracing

Predict variable values and output by tracing a program step by step; show the effect of `break` and `continue`.

Control Structures

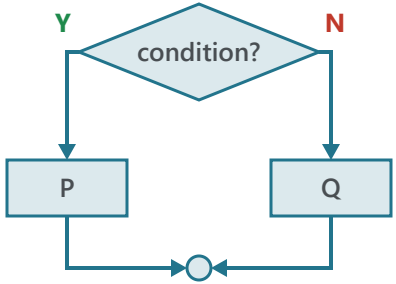
- **Control structures** direct a program's **flow**: which statements run, in what order, how often.
- Java has three basic control structures; any program can be built from them:

Sequence



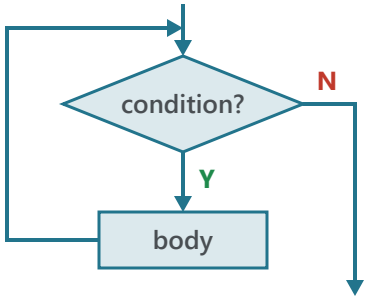
Statements run top to bottom, once each.

Decisions



A condition picks one of two (or more) paths: `if-else`, `switch`.

Loops



Work repeats while a condition holds: `for`, `while`, `do-while`.

Why it matters: decisions let a program react to different situations and user input; loops automate repetitive work.

Sequential Execution

Code runs **top to bottom, line by line**; one operation finishes before the next starts. A program with no control structures is sequential.

```
▶ int x = 5;
2 int y = 10;
3 int result = x + y;
4 System.out.println(result);
```



The first line runs: `x` gets 5. The next line does not start **until this one finishes**.

Step 1 / 4

VARIABLES

x = 5

OUTPUT

Sequential Execution

Code runs **top to bottom, line by line**; one operation finishes before the next starts. A program with no control structures is sequential.

```
1 int x = 5;
▶ int y = 10;
3 int result = x + y;
4 System.out.println(result);
```



Second line: `y` gets 10.

Step 2 / 4

VARIABLES

`x = 5` `y = 10`

OUTPUT

Sequential Execution

Code runs **top to bottom, line by line**; one operation finishes before the next starts. A program with no control structures is sequential.

```
1 int x = 5;
2 int y = 10;
▶ int result = x + y;
4 System.out.println(result);
```



The third line uses the results of the first two: 5 + 10 = 15.

Step 3 / 4

VARIABLES

`x = 5` `y = 10` `result = 15`

OUTPUT

Sequential Execution

Code runs **top to bottom, line by line**; one operation finishes before the next starts. A program with no control structures is sequential.

```
1 int x = 5;
2 int y = 10;
3 int result = x + y;
▶ System.out.println(result);
```



The last line prints the result. If the lines were reordered (e.g. line 3 first), the program would not compile.

Step 4 / 4

VARIABLES

`x = 5` `y = 10` `result = 15`

OUTPUT

```
15
```

Making Decisions

- **Decision making** means the program chooses which path to follow **at run time**.
- It lets the program carry out different actions depending on **conditions**.
- In Java this is done with `if-else` and `switch-case`.
- A condition is a `true`- or `false`-valued **boolean expression**.
- Analogy: a sign at a fork in the road. “If it's raining, take an umbrella; otherwise don't.”
- Why it matters: the program responds to different situations and user input, the flow is steered, and flexibility increases.

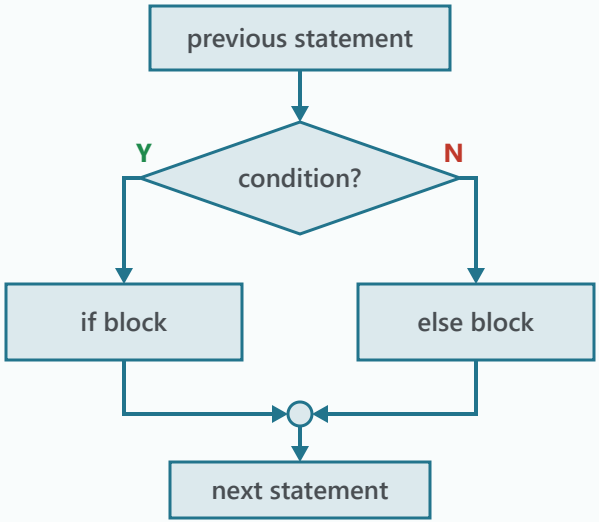
Operator	Meaning	Example (x = 7)
`==`	equal	`x == 7` → true
`!=`	not equal	`x != 7` → false
`<` `>`	less / greater	`x < 10` → true
`<=` `>=`	less-or-equal / greater-or-equal	`x >= 8` → false

`=` is assignment, `==` is comparison. `if (x = 7)` does not compile: the condition must be a `boolean`.

The if-else Statement

```
if (condition) {
    // runs if the condition is true
} else {
    // runs if the condition is false
}
```

- First the **condition is evaluated**.
- If the condition is **true** (`true`), the `if` block runs and the `else` block is **skipped**.
- If the condition is **false** (`false`), the `if` block is skipped and the `else` block runs.
- The `else` is **optional**. Of the two blocks, at most **one** runs.
- The condition always goes in parentheses: `if (condition)` .



if Alone, and Braces

ELSE-LESS IF

```
int temp = 35;
if (temp > 30) {
    System.out.println("It's hot!");
}
System.out.println("The program continues.");
```

```
It's hot!
The program continues.
```

If the condition were false (e.g. `temp = 20`), only "The program continues." would print. After `if`, the next line **always** runs.

PITFALL: NO BRACES

```
int temp = 35;
if (temp > 40)
    System.out.println("Very hot!");
    System.out.println("Drink water.");
```

```
Drink water.
```

Without braces, `if` owns **only the first statement**. Indentation means nothing to Java: "Drink water." runs regardless of the condition. That's why braces `{ }` are used even for one-line blocks.

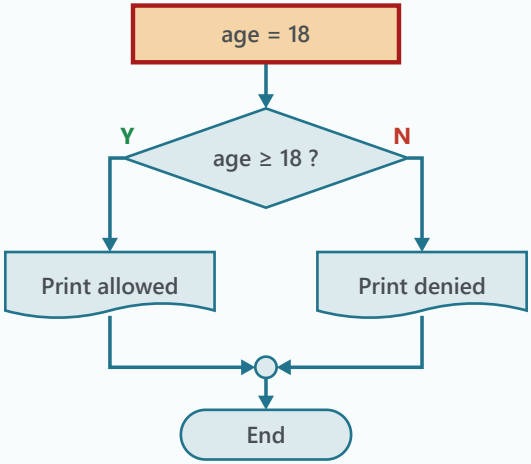
Example: Age Check

```
1 int age = 18;
2 if (age >= 18) {
3     System.out.println("You can get a license.");
4 } else {
5     System.out.println("You cannot get a license.");
6 }
```

VARIABLES

`age = 18`

OUTPUT



`age` is assigned 18.

Step 1 / 4

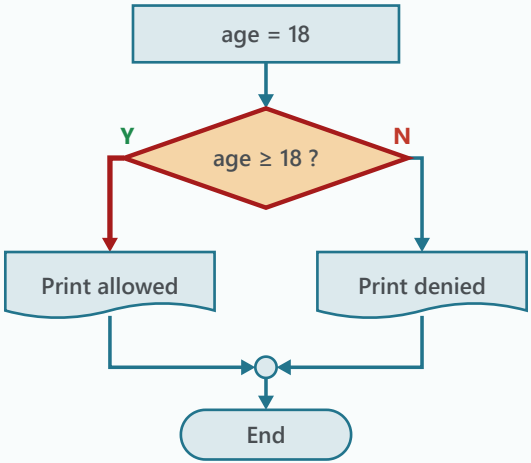
Example: Age Check

```
1 int age = 18;
▶ if (age >= 18) {
3     System.out.println("You can get a license.");
4 } else {
5     System.out.println("You cannot get a license.");
6 }
```

VARIABLES

`age = 18`

OUTPUT



The condition is evaluated: 18 >= 18 → **true**. Flow takes the “Y” (yes) branch.

Example: Age Check

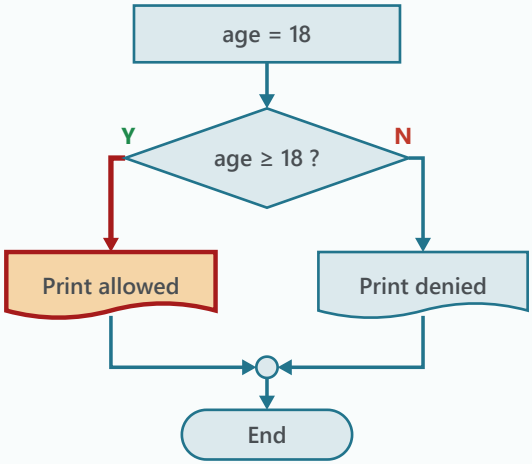
```
1 int age = 18;
2 if (age >= 18) {
▶     System.out.println("You can get a license.");
4 } else {
5     System.out.println("You cannot get a license.");
6 }
```

VARIABLES

`age = 18`

OUTPUT

```
You can get a license.
```



The `if` block runs and prints the message.

Example: Age Check

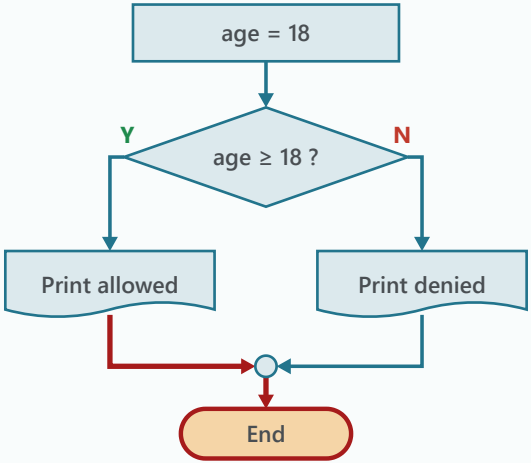
```
1 int age = 18;
2 if (age >= 18) {
3     System.out.println("You can get a license.");
4 } else {
5     System.out.println("You cannot get a license.");
▶ }
```

VARIABLES

`age = 18`

OUTPUT

```
You can get a license.
```



The `else` block (line 5) is **skipped**. `age = 16` would make it false, and only “You cannot get a license.” would be printed.

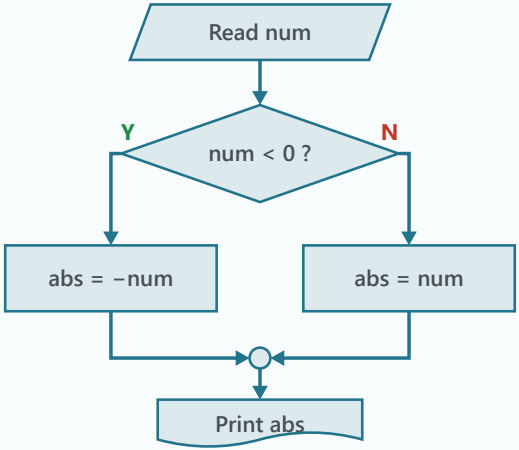
Example: Absolute Value

```
▶ Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a number: ");
3 double num = scanner.nextDouble();
4 double absValue;
5 if (num < 0) {
6     absValue = -num;
7 } else {
8     absValue = num;
9 }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

—

OUTPUT



To read from the keyboard, create a `Scanner` object (this needs `import java.util.Scanner;` at the top of the file).

Example: Absolute Value

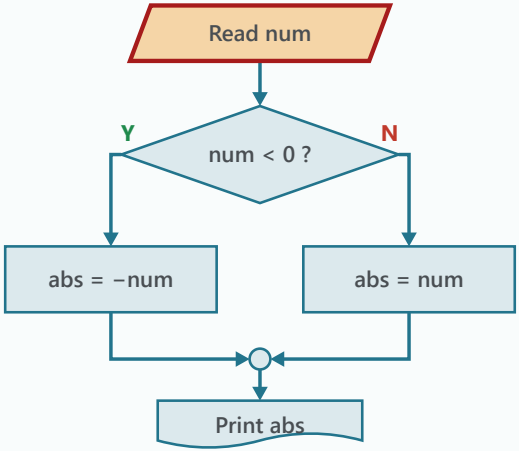
```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter a number: ");
3  double num = scanner.nextDouble();
4  double absValue;
5  if (num < 0) {
6      absValue = -num;
7  } else {
8      absValue = num;
9  }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

—

OUTPUT

```
Enter a number:
```



The user is told what to enter. `print` does not move to a new line; the cursor waits on the same line.

Example: Absolute Value

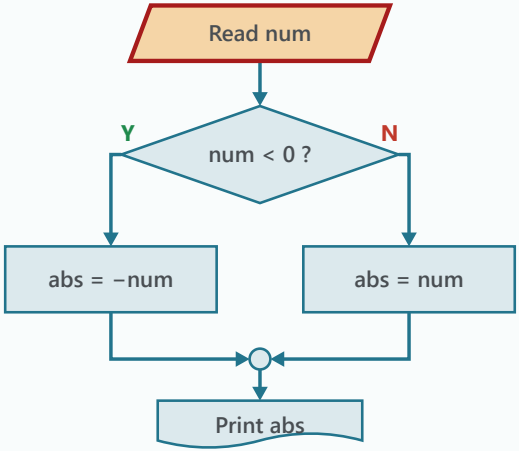
```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter a number: ");
▶  double num = scanner.nextDouble();
4  double absValue;
5  if (num < 0) {
6      absValue = -num;
7  } else {
8      absValue = num;
9  }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

`num = -7.5`

OUTPUT

```
Enter a number: -7,5
```



The user types **-7,5**. On a computer with a Turkish locale the decimal separator is a comma, so `-7.5` throws `InputMismatchException`.

Example: Absolute Value

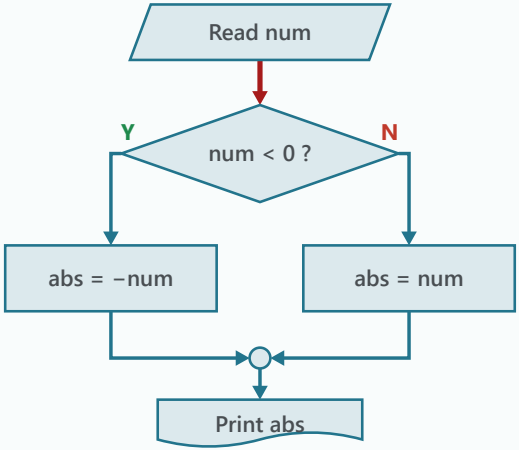
```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter a number: ");
3  double num = scanner.nextDouble();
▶  double absValue;
5  if (num < 0) {
6      absValue = -num;
7  } else {
8      absValue = num;
9  }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

`num = -7.5` `absValue = ?`

OUTPUT

```
Enter a number: -7,5
```



`absValue` is declared but has no value yet.

Example: Absolute Value

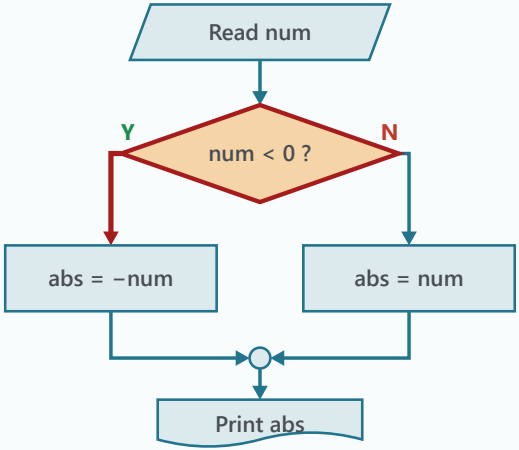
```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter a number: ");
3  double num = scanner.nextDouble();
4  double absValue;
▶  if (num < 0) {
6      absValue = -num;
7  } else {
8      absValue = num;
9  }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

`num = −7.5` `absValue = ?`

OUTPUT

```
Enter a number: -7,5
```



Condition: -7.5 < 0 → **true**.

Example: Absolute Value

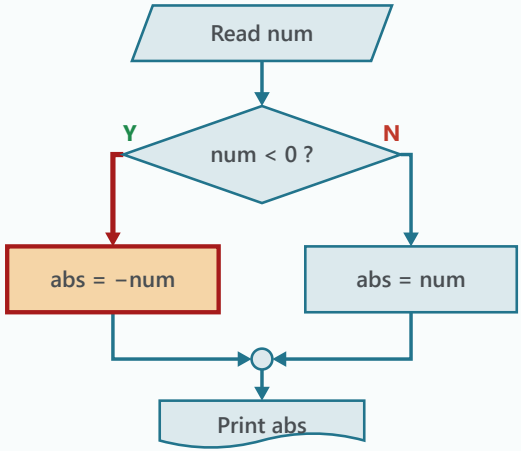
```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter a number: ");
3  double num = scanner.nextDouble();
4  double absValue;
5  if (num < 0) {
▶      absValue = -num;
7  } else {
8      absValue = num;
9  }
10 System.out.println("Absolute value: " + absValue);
```

VARIABLES

`num = -7.5` `absValue = 7.5`

OUTPUT

```
Enter a number: -7,5
```



`-num` = −(−7.5) = 7.5 is assigned. The `else` block is skipped.

Step 6 / 7

Example: Absolute Value

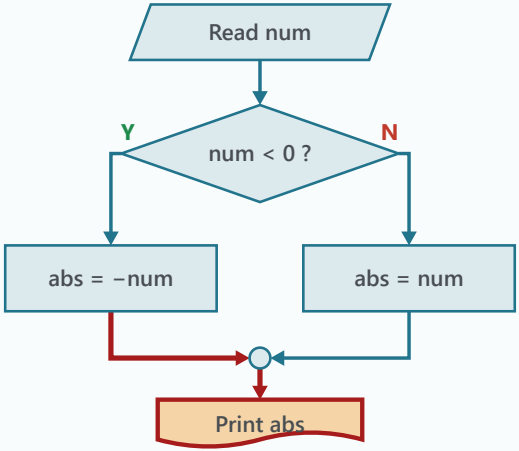
```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter a number: ");
3 double num = scanner.nextDouble();
4 double absValue;
5 if (num < 0) {
6     absValue = -num;
7 } else {
8     absValue = num;
9 }
▶ System.out.println("Absolute value: " + absValue);
```

VARIABLES

`num = -7.5` `absValue = 7.5`

OUTPUT

```
Enter a number: -7,5
Absolute value: 7.5
```



The result is printed. `println`, however, always prints a `double` value with a dot: 7.5.

else-if Chains

```java
import java.util.Scanner;

public class PositiveNegative {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        double num = scanner.nextDouble();
        if (num > 0) {
            System.out.println("Positive.");
        } else if (num < 0) {
            System.out.println("Negative.");
        } else {
            System.out.println("Zero.");
        }
    }
}
```

- With more than two options, `else if` tests the conditions **one after another**.
- Conditions are tried **top to bottom**; the block of the **first true** condition runs, and the rest are skipped.
- If none is true, the final `else` runs (if present).

THREE SEPARATE RUNS

```
Enter a number: 4
Positive.
```

```
Enter a number: -2
Negative.
```

```
Enter a number: 0
Zero.
```

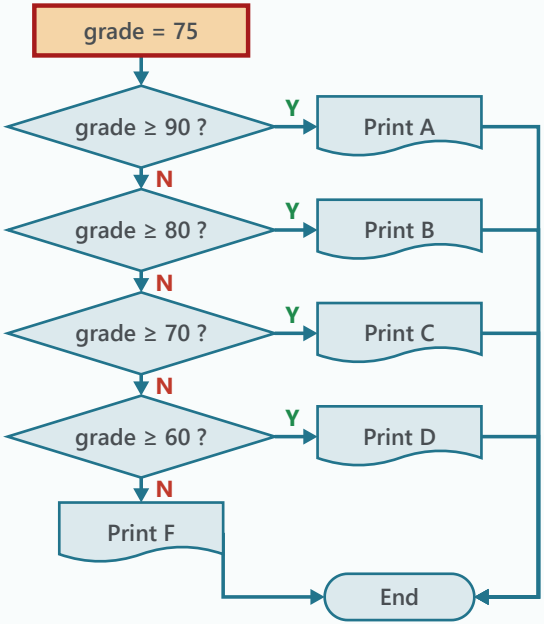
else-if Chain: Letter Grade

```
▶ int grade = 75;
2 if (grade >= 90) {
3     System.out.println("A");
4 } else if (grade >= 80) {
5     System.out.println("B");
6 } else if (grade >= 70) {
7     System.out.println("C");
8 } else if (grade >= 60) {
9     System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT



`grade` = 75.

Step 1 / 6

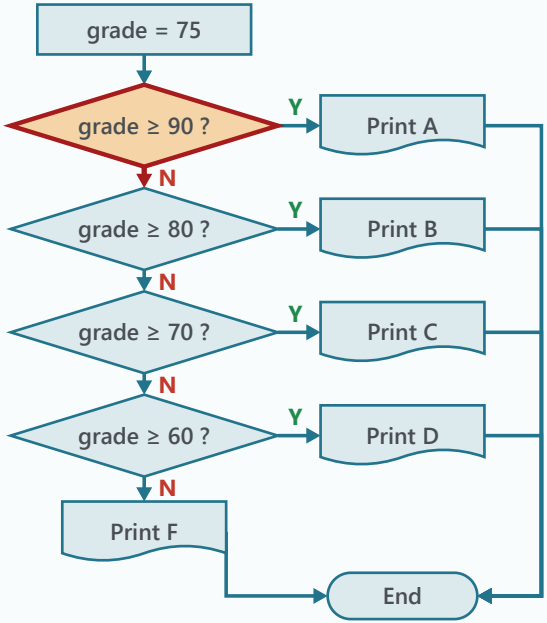
else-if Chain: Letter Grade

```
1  int grade = 75;
▶  if (grade >= 90) {
3      System.out.println("A");
4  } else if (grade >= 80) {
5      System.out.println("B");
6  } else if (grade >= 70) {
7      System.out.println("C");
8  } else if (grade >= 60) {
9      System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT



75 >= 90 → false. On to the next `else if`.

Step 2 / 6

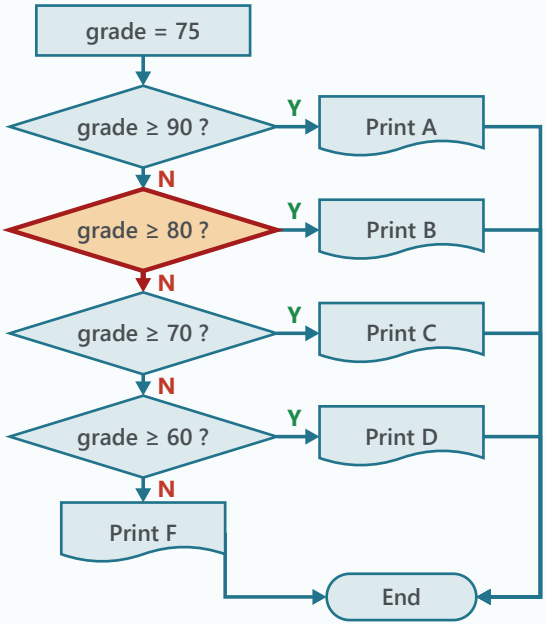
else-if Chain: Letter Grade

```
1  int grade = 75;
2  if (grade >= 90) {
3      System.out.println("A");
▶  } else if (grade >= 80) {
5      System.out.println("B");
6  } else if (grade >= 70) {
7      System.out.println("C");
8  } else if (grade >= 60) {
9      System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT



75 >= 80 → false. On to the next condition.

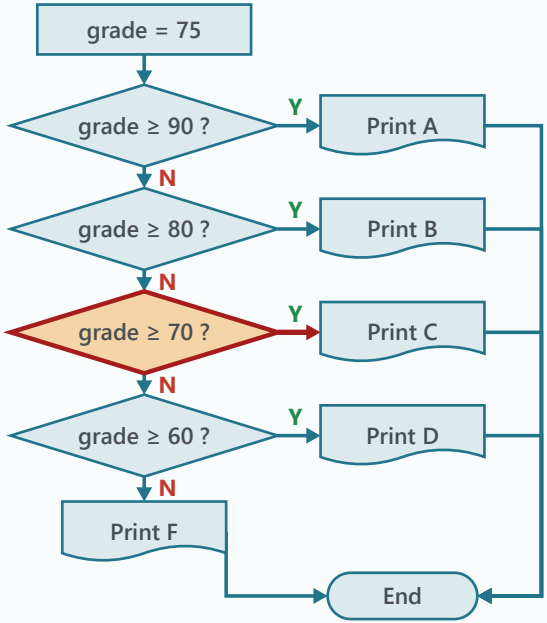
else-if Chain: Letter Grade

```
1  int grade = 75;
2  if (grade >= 90) {
3      System.out.println("A");
4  } else if (grade >= 80) {
5      System.out.println("B");
▶  } else if (grade >= 70) {
7      System.out.println("C");
8  } else if (grade >= 60) {
9      System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT



75 >= 70 → **true**: the first true condition is found.

else-if Chain: Letter Grade

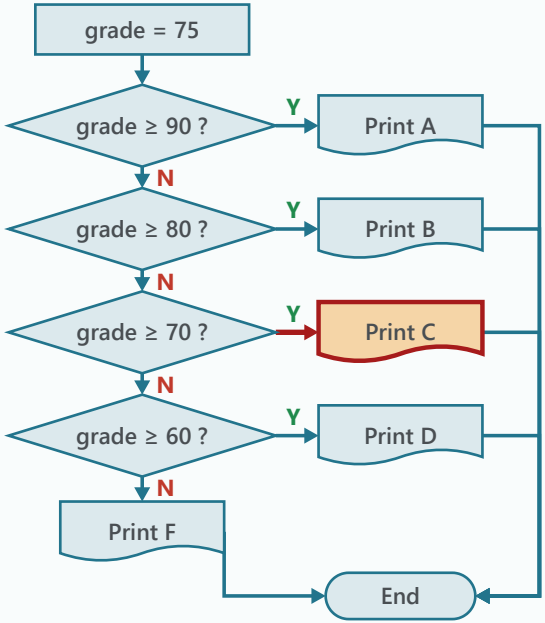
```
1  int grade = 75;
2  if (grade >= 90) {
3      System.out.println("A");
4  } else if (grade >= 80) {
5      System.out.println("B");
6  } else if (grade >= 70) {
▶      System.out.println("C");
8  } else if (grade >= 60) {
9      System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT

```
C
```



"C" is printed.

else-if Chain: Letter Grade

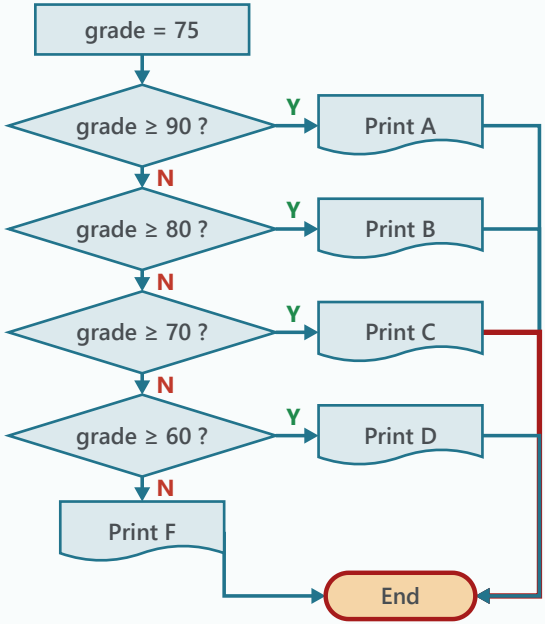
```
1  int grade = 75;
2  if (grade >= 90) {
3      System.out.println("A");
4  } else if (grade >= 80) {
5      System.out.println("B");
6  } else if (grade >= 70) {
7      System.out.println("C");
8  } else if (grade >= 60) {
9      System.out.println("D");
10 } else {
11     System.out.println("F");
12 }
```

VARIABLES

`grade = 75`

OUTPUT

```
C
```

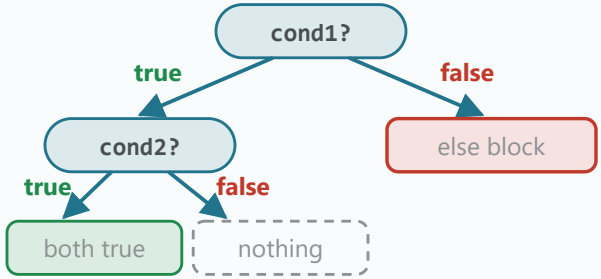


The remaining conditions (including `grade >= 60`) **are never tested**. Order matters: if `>= 60` were at the top, 75 would get a “D”.

Nested Decisions

- Inside the block of one decision you can place **another decision**.
- Useful for more complex conditions: once the outer condition holds, a second question is asked.
- The inner `if` is tested only if the outer condition is **true**.
- Deep nesting **hurts readability**; indent properly and, where possible, flatten with `&&` or `else if`: `if (age >= 18) { if (hasLicense) … }` and `if (age >= 18 && hasLicense) …` do the same thing.
- Range checks are also written with `&&`: `grade >= 60 && grade < 90`; `60 <= grade < 90` does not compile.

```
if (cond1) {
    if (cond2) {
        // cond1 AND cond2 true
    }
} else {
    // cond1 false
}
```



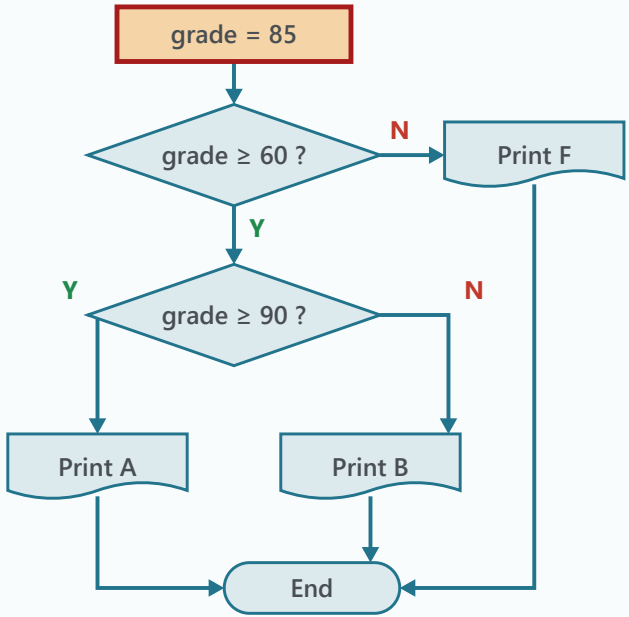
Nested Decision: Letter Grade

```
1 int grade = 85;
2 if (grade >= 60) {
3     if (grade >= 90) {
4         System.out.println("Letter grade: A");
5     } else {
6         System.out.println("Letter grade: B");
7     }
8 } else {
9     System.out.println("Letter grade: F");
10 }
```

VARIABLES

`grade = 85`

OUTPUT



`grade` = 85.

Step 1 / 5

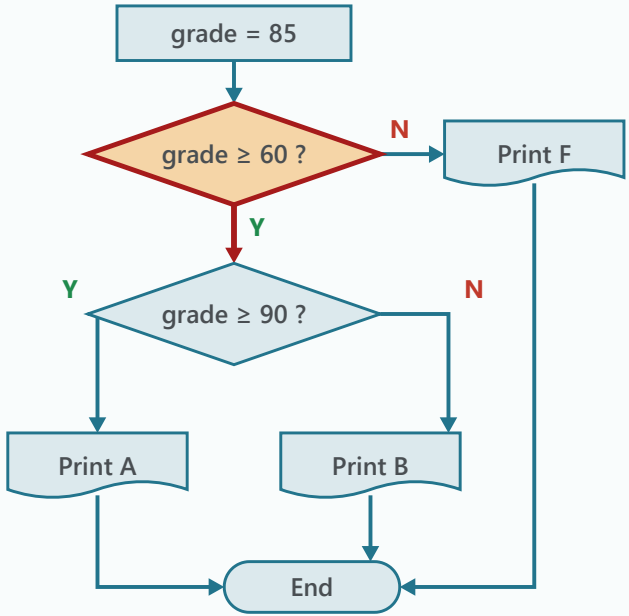
Nested Decision: Letter Grade

```
1  int grade = 85;
▶ if (grade >= 60) {
3      if (grade >= 90) {
4          System.out.println("Letter grade:
   A");
5      } else {
6          System.out.println("Letter grade:
   B");
7      }
8  } else {
9      System.out.println("Letter grade: F");
10 }
```

VARIABLES

`grade = 85`

OUTPUT



Outer condition: 85 >= 60 → **true**. Enter the outer `if` block.

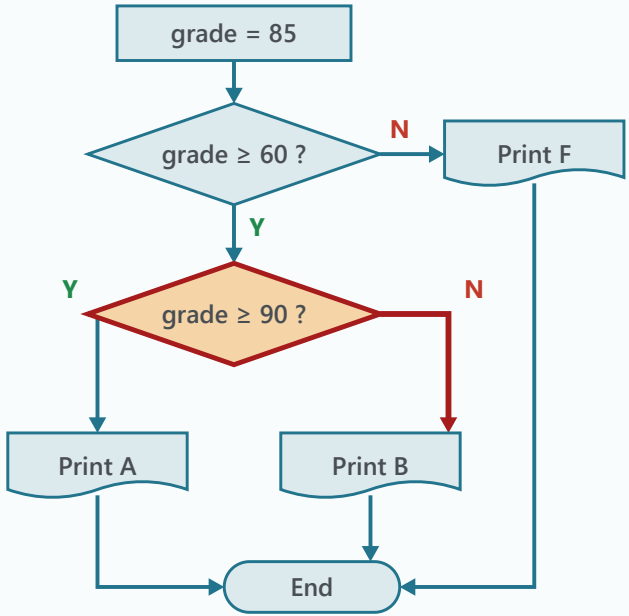
Nested Decision: Letter Grade

```
1  int grade = 85;
2  if (grade >= 60) {
▶      if (grade >= 90) {
4          System.out.println("Letter grade:
   A");
5      } else {
6          System.out.println("Letter grade:
   B");
7      }
8  } else {
9      System.out.println("Letter grade: F");
10 }
```

VARIABLES

`grade = 85`

OUTPUT



Inner condition: 85 >= 90 → false. Go to the inner `else`.

Nested Decision: Letter Grade

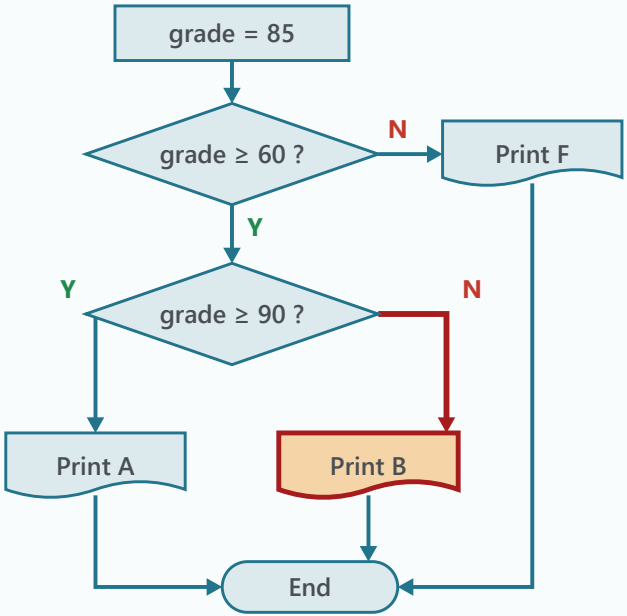
```
1  int grade = 85;
2  if (grade >= 60) {
3      if (grade >= 90) {
4          System.out.println("Letter grade:
   A");
5      } else {
▶          System.out.println("Letter grade:
   B");
7      }
8  } else {
9      System.out.println("Letter grade: F");
10 }
```

VARIABLES

`grade = 85`

OUTPUT

```
Letter grade: B
```



"Letter grade: B" is printed.

Nested Decision: Letter Grade

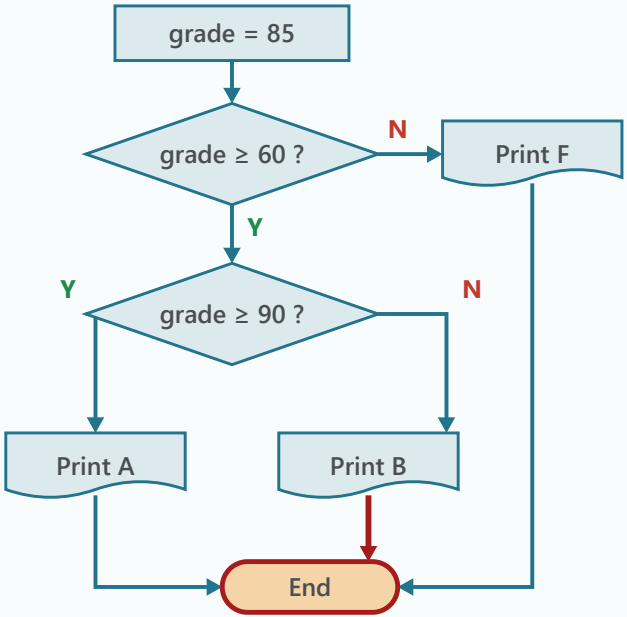
```
1 int grade = 85;
2 if (grade >= 60) {
3     if (grade >= 90) {
4         System.out.println("Letter grade:
  A");
5     } else {
6         System.out.println("Letter grade:
  B");
7     }
8 } else {
9     System.out.println("Letter grade: F");
▶ }
```

VARIABLES

`grade = 85`

OUTPUT

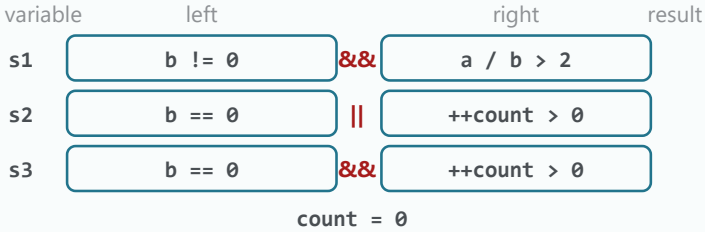
```
Letter grade: B
```



The outer `else` (line 9) is skipped. `grade = 45` would skip the inner `if` entirely and print F directly.

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



`b = 0` : `a / b` would cause a division-by-zero error (`ArithmeticException`) if evaluated. `count` will count whether the right-hand sides run.

Step 1 / 9

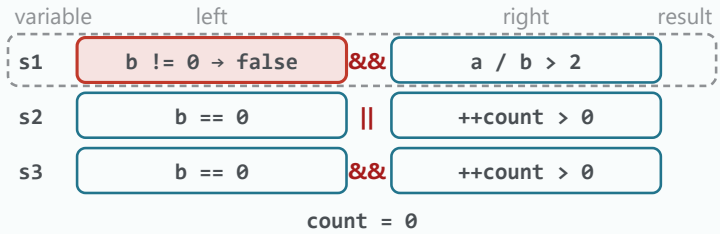
VARIABLES

`a = 10` `b = 0` `count = 0`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
▶ boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



`&&` first evaluates the **left** side: `b != 0` → false.

Step 2 / 9

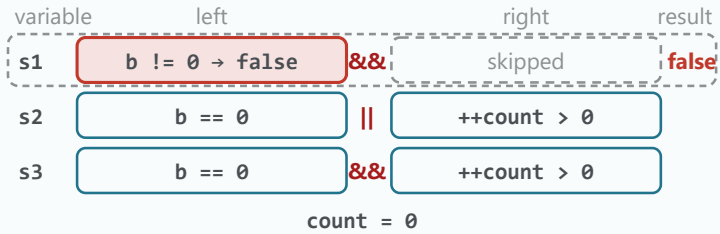
VARIABLES

`a = 10` `b = 0` `count = 0`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
▶ boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



If the left is false, `&&` 's result is already false: `a / b > 2` **is never evaluated**, so no error occurs. `s1 = false` .

Step 3 / 9

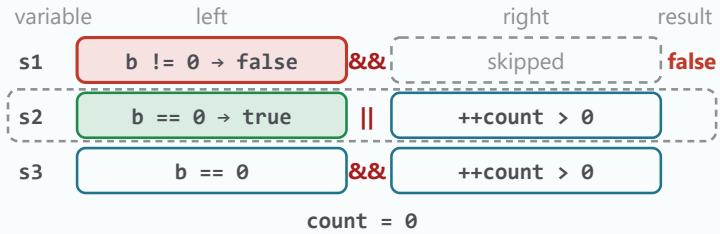
VARIABLES

`a = 10` `b = 0` `count = 0` `s1 = false`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
▶ boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



`||` : left side `b == 0` → true.

Step 4 / 9

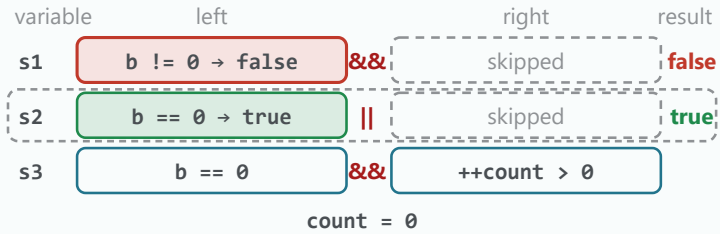
VARIABLES

`a = 10` `b = 0` `count = 0` `s1 = false`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
▶ boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



If the left is true, `||` 's result is already true: `++count` **does not run**, `count` stays 0. `s2 = true` .

Step 5 / 9

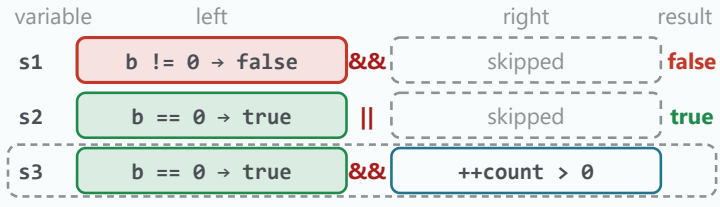
VARIABLES

`a = 10` `b = 0` `count = 0` `s1 = false` `s2 = true`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
▶ boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



Again `&&` ; the left side `b == 0` → true. The result is not yet known.

Step 6 / 9

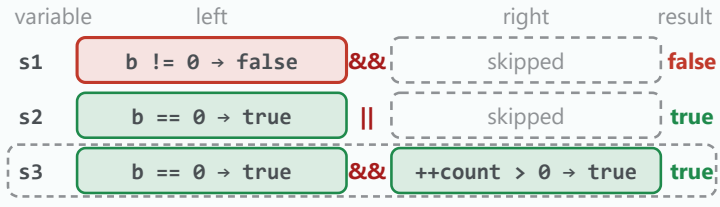
VARIABLES

`a = 10` `b = 0` `count = 0` `s1 = false` `s2 = true`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
▶ boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



The right side **must be evaluated**: `++count` → 1, `1 > 0` → true. `s3 = true`.

Step 7 / 9

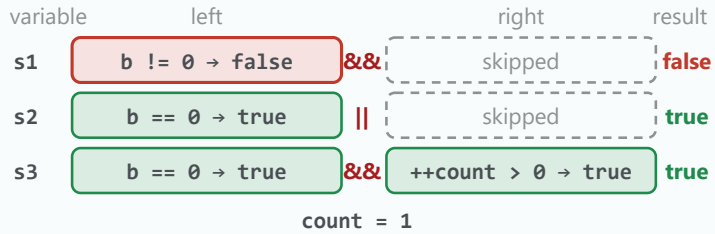
VARIABLES

`a = 10` `b = 0` `count = 1` `s1 = false` `s2 = true` `s3 = true`

OUTPUT

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
▶ System.out.println(s1 + " " + s2 + " " +
  s3);
6 System.out.println("count = " + count);
```



The three results are printed: `false true true`.

Step 8 / 9

VARIABLES

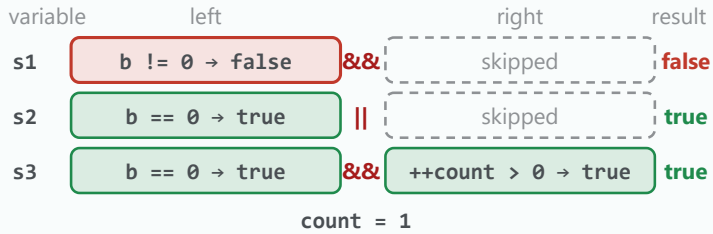
`a = 10` `b = 0` `count = 1` `s1 = false` `s2 = true` `s3 = true`

OUTPUT

```
false true true
```

Short-Circuit: Tracing the Code

```
1 int a = 10, b = 0, count = 0;
2 boolean s1 = b != 0 && a / b > 2;
3 boolean s2 = b == 0 || ++count > 0;
4 boolean s3 = b == 0 && ++count > 0;
5 System.out.println(s1 + " " + s2 + " " +
  s3);
▶ System.out.println("count = " + count);
```



`count = 1` : only one of the three right-hand sides ran. Side effects (`++`) on the right of `&&` / `||` are therefore misleading.

Step 9 / 9

VARIABLES

`a = 10` `b = 0` `count = 1` `s1 = false` `s2 = true` `s3 = true`

OUTPUT

```
false true true
count = 1
```

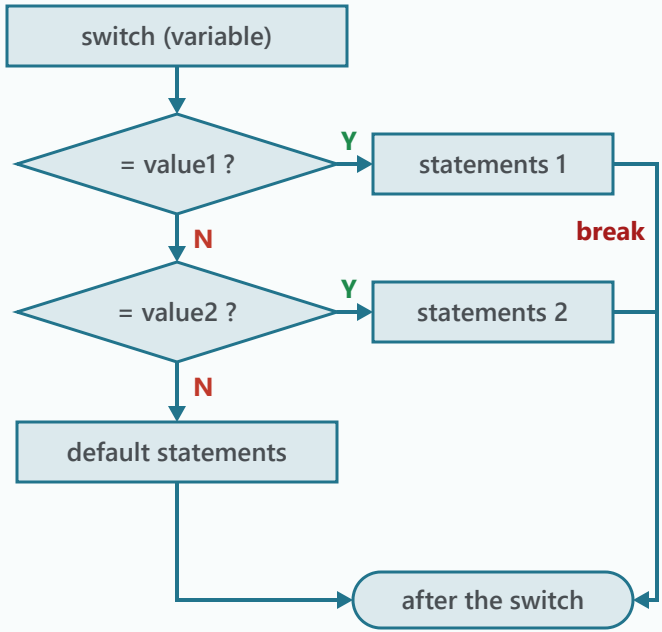
The switch-case Statement

- **switch-case** compares the value of a variable against **a set of constant** values and acts according to the match.
- It is used to choose **one path** among several options.
- When the same variable gets many `==` comparisons, it is tidier than an `else if` chain.
- Analogy: elevator buttons. The floor number you press picks where you go.

```
switch (variable) {  // value being checked
    case value1:     // one value per case
        // statements 1
        break;
    case value2:
        // statements 2
        break;
    // ...
    default:         // if nothing matches
        // default statements (optional)
}
```

How switch-case Works

- The variable is compared with the `case` values **top to bottom**.
- If a matching `case` is found, that `case` 's statements run, and then `break` exits the `switch` .
- If no `case` matches, the `default` section (if any) runs.
- `default` absent as well? Then the `switch` exits without doing anything.
- `break` missing? Execution **falls into the next case** (fall-through).



Example: Finding the Season

```
▶ int month = 4;
2 String season;
3 switch (month) {
4     case 3: case 4: case 5:
5         season = "Spring";
6         break;
7     case 6: case 7: case 8:
8         season = "Summer";
9         break;
10    case 9: case 10: case 11:
11        season = "Autumn";
12        break;
13    case 12: case 1: case 2:
14        season = "Winter";
15        break;
16    default:
17        season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```

3 4 5	→ Spring
6 7 8	→ Summer
9 10 11	→ Autumn
12 1 2	→ Winter
default	→ Unknown

`month` = 4.

Step 1 / 7

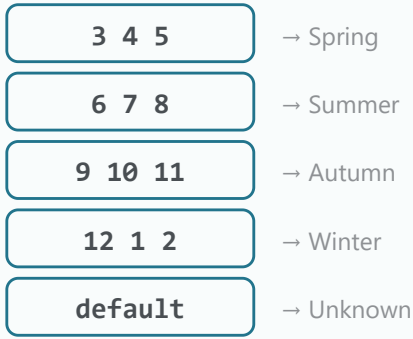
VARIABLES

`month = 4`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
3  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```



`season` is declared, with no value yet.

Step 2 / 7

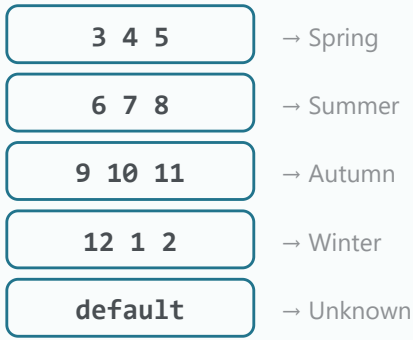
VARIABLES

`month = 4` `season = ?`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
▶  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```



`switch (month)` : the value 4 is computed once, then the `case` labels are searched.

Step 3 / 7

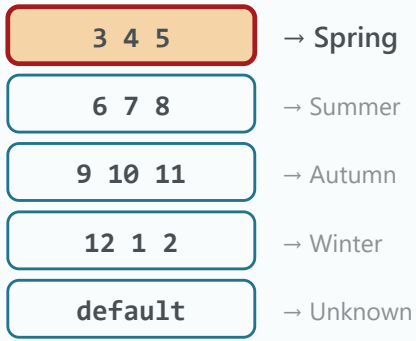
VARIABLES

`month = 4` `season = ?`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
3  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```



`case 4` matches. The three `case` labels on that line lead to the same code.

Step 4 / 7

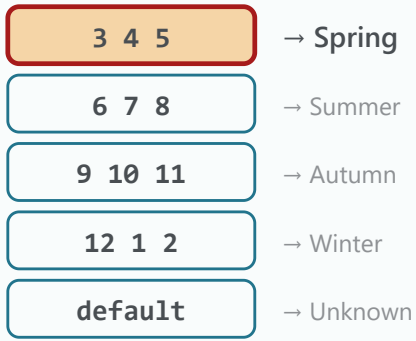
VARIABLES

`month = 4` `season = ?`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
3  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```



`season` = "Spring".

Step 5 / 7

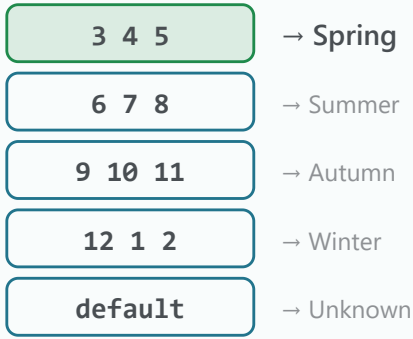
VARIABLES

`month = 4` `season = Spring`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
3  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
19 System.out.println(month + ": " + season);
```



`break` : `switch` ends; execution jumps past its closing brace (line 18).

Step 6 / 7

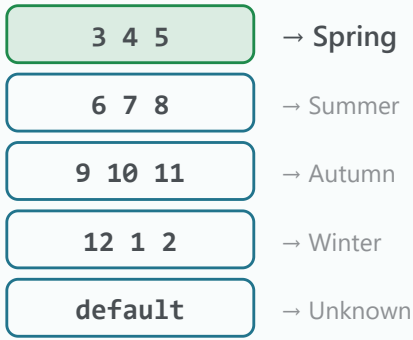
VARIABLES

`month = 4` `season = Spring`

OUTPUT

Example: Finding the Season

```
1  int month = 4;
2  String season;
3  switch (month) {
4      case 3: case 4: case 5:
5          season = "Spring";
6          break;
7      case 6: case 7: case 8:
8          season = "Summer";
9          break;
10     case 9: case 10: case 11:
11         season = "Autumn";
12         break;
13     case 12: case 1: case 2:
14         season = "Winter";
15         break;
16     default:
17         season = "Unknown";
18 }
▶  System.out.println(month + ": " + season);
```



`default` guarantees that `season` gets a value on every path; without it: *variable season might not have been initialized*.

Step 7 / 7

VARIABLES

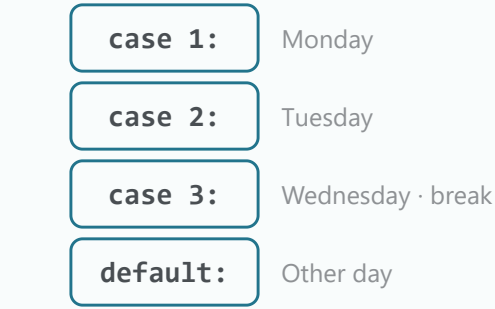
`month = 4` `season = Spring`

OUTPUT

```
4: Spring
```

Forgetting break: Fall-Through

```
▶ int day = 2;
2 switch (day) {
3     case 1:
4         System.out.println("Monday");
5     case 2:
6         System.out.println("Tuesday");
7     case 3:
8         System.out.println("Wednesday");
9         break;
10    default:
11        System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



`day` = 2.

Step 1 / 8

VARIABLES

`day` = **2**

OUTPUT

Forgetting break: Fall-Through

```
1  int day = 2;
▶  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
7      case 3:
8          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```

`case 1:`	Monday
`case 2:`	Tuesday
`case 3:`	Wednesday · break
`default:`	Other day

The `case` matching 2 is searched for.

Step 2 / 8

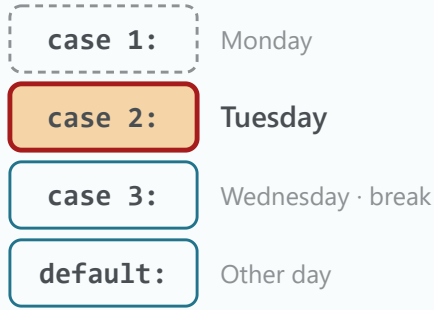
VARIABLES

`day = 2`

OUTPUT

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
▶      case 2:
6          System.out.println("Tuesday");
7      case 3:
8          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



`case 1` is skipped, `case 2` matches.

Step 3 / 8

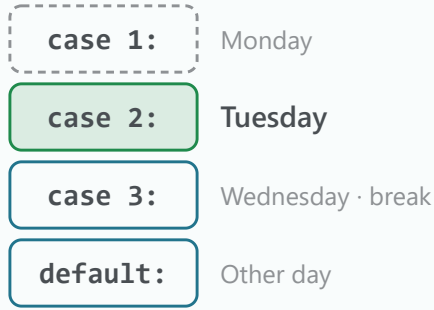
VARIABLES

`day = 2`

OUTPUT

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
7      case 3:
8          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



"Tuesday" is printed. There is no `break` **here!**

Step 4 / 8

VARIABLES

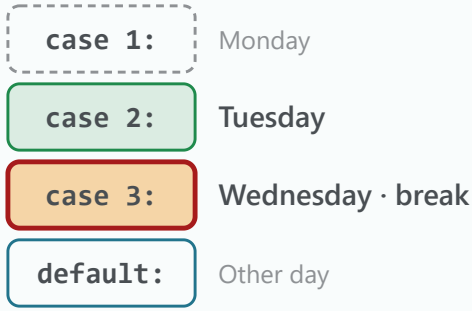
`day = 2`

OUTPUT

```
Tuesday
```

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
▶      case 3:
8          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



The `case 3:` label is not a **test**; execution falls through to the code below.

Step 5 / 8

VARIABLES

`day = 2`

OUTPUT

```
Tuesday
```

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
7      case 3:
▶          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



"Wednesday" is printed too, even though the day is not 3.

VARIABLES

`day = 2`

OUTPUT

```
Tuesday
Wednesday
```

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
7      case 3:
8          System.out.println("Wednesday");
▶          break;
10     default:
11         System.out.println("Other day");
12 }
13 System.out.println("switch done");
```



`break` exits the `switch`; `default` does not run.

Step 7 / 8

VARIABLES

`day = 2`

OUTPUT

```
Tuesday
Wednesday
```

Forgetting break: Fall-Through

```
1  int day = 2;
2  switch (day) {
3      case 1:
4          System.out.println("Monday");
5      case 2:
6          System.out.println("Tuesday");
7      case 3:
8          System.out.println("Wednesday");
9          break;
10     default:
11         System.out.println("Other day");
12 }
▶ System.out.println("switch done");
```



Execution continues after the `switch`. To avoid fall-through, end every `case` with `break`.

Step 8 / 8

VARIABLES

`day = 2`

OUTPUT

```
Tuesday
Wednesday
switch done
```

switch: char and String

CHAR-BASED SIMPLE CALCULATOR

```
int a = 12, b = 4;
char op = '*';
int result;
switch (op) {
    case '+':  result = a + b;  break;
    case '-':  result = a - b;  break;
    case '*':  result = a * b;  break;
    case '/':  result = a / b;  break;
    default:
        System.out.println("Bad operator");
        result = 0;
}
System.out.println(a + " " + op + " " + b
        + " = " + result);
```

```
12 * 4 = 48
```

STRING-BASED COMMAND SELECTION (JAVA 7+)

```
String command = "stop";
switch (command) {
    case "start":
        System.out.println("Engine started");
        break;
    case "stop":
        System.out.println("Engine stopped");
        break;
    default:
        System.out.println("Unknown command");
}
```

```
Engine stopped
```

Short `case` s can go on a single line: `case '+': result = a + b; break;` . `String` comparison is case-sensitive: `"Stop"` ≠ `"stop"` .

switch Rules

- The selector (the expression in parentheses) can be: `byte`, `short`, `char`, `int` (and their wrappers), `String`, `enum`.
- `long`, `float`, `double`, `boolean` cannot be used with a classic `switch`.
- `case` values **must be compile-time constants** (`5`, `'a'`, `"stop"`, `final` constants).
- The same value cannot appear in two `case`s.
- Ranges (`month >= 3 && month <= 5`) or complex conditions need `if-else`.

```
double d = 2.5;
switch (d) { ... }
```

```
error: selector type double is not allowed
```

```
int b = 2;
switch (a) { case b: break; }
```

```
error: constant expression required
```

```
switch (a) { case 1: break; case 1: break; }
```

```
error: duplicate case label
```

switch-case vs. if-else

	`if-else / else if`	`switch-case`
What is tested?	Any `boolean` condition: ranges, compound conditions, different variables	Equality of a single expression with **constant** values
Flexibility	More flexible	Only certain types and constant `case` values
With many options	Long chains can look messy	A neat, readable list
Pitfall	Conditions in the wrong order pick the wrong branch	`break` missing → falls into the cases below
Example	`if (grade >= 90) … else if (grade >= 80) …`	`switch (month) { case 3: … }`

Every `switch` can be written as an `else if` chain; the reverse is not always possible (e.g. ranges like `grade >= 90`).

The New switch (Java 14+)

```
int month = 11;
String season = switch (month) {
    case 3, 4, 5 -> "Spring";
    case 6, 7, 8 -> "Summer";
    case 9, 10, 11 -> "Autumn";
    case 12, 1, 2 -> "Winter";
    default -> "Unknown";
};
System.out.println(season);
```

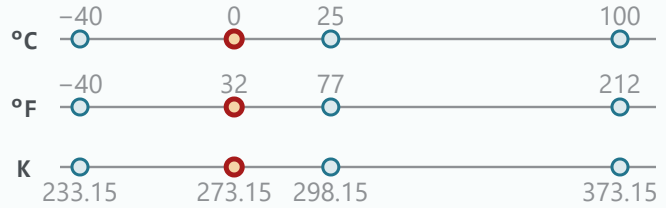
```
Autumn
```

- Java 14 introduced the **arrow form**: `case 3, 4, 5 -> …`
- A `case` can take several values separated by commas.
- The arrow form **has no fall-through**; no `break` is needed.
- **A switch expression** produces a value that can be assigned directly; it ends with `;`.
- The expression form must cover every possibility: for `int`, `default` is required.
- This course uses the classic form; the arrow form is available from Java 14 on.

APPLICATION: TEMPERATURE

Temperature Conversion: Table and Formulas

Celsius (°C)	Fahrenheit (°F)	Kelvin (K)
−40	−40	233.15
0	32	273.15
25	77	298.15
100	212	373.15



Celsius → Fahrenheit

```
°F = (°C × 1.8) + 32
```

Fahrenheit → Celsius

```
°C = (°F − 32) / 1.8
```

Celsius → Kelvin

```
K = °C + 273.15
```

Goal: translate the formulas into Java and compute them for one value.

APPLICATION: TEMPERATURE

Temperature Conversion: One Value

```java
public class Program {
    public static void main(String[] args) {
        double celsius = 25.0;
        double fahrenheit = (celsius * 1.8) + 32;
        double kelvin = celsius + 273.15;
        System.out.println(celsius + " °C = " + fahrenheit + " °F = " + kelvin + " K");
    }
}
```

```
25.0 °C = 77.0 °F = 298.15 K
```

- A sequential program: three lines of computation, one line of output; no decisions or loops.
- Use `double` for decimal values. Parentheses make the order explicit: first `celsius * 1.8`, then `+ 32`.
- `+` used with a string performs **concatenation**; `double` values print as `25.0`.
- Every computed value should be used: had `kelvin` not been printed, computing it would have been wasted.

Loops and Their Kinds

A loop repeats the same operation **several times**. Analogy: running 10 laps on a track; every lap has the same moves, and the lap counter goes up by one.

for

When the number of repetitions is known; works over a range of values.

```
for (int i = 0; i < 5; i++) { … }
```

while

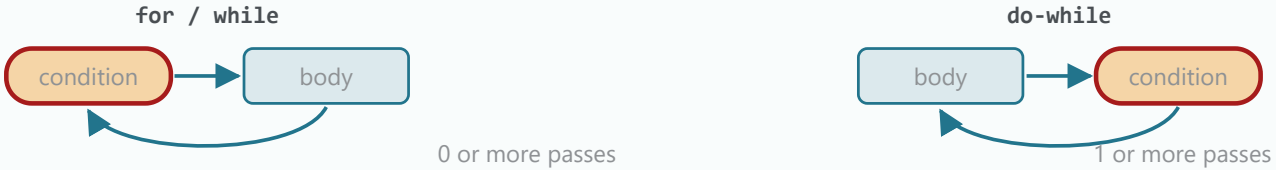
Runs while a condition holds; the number of repetitions may not be known in advance.

```
while (condition) { … }
```

do-while

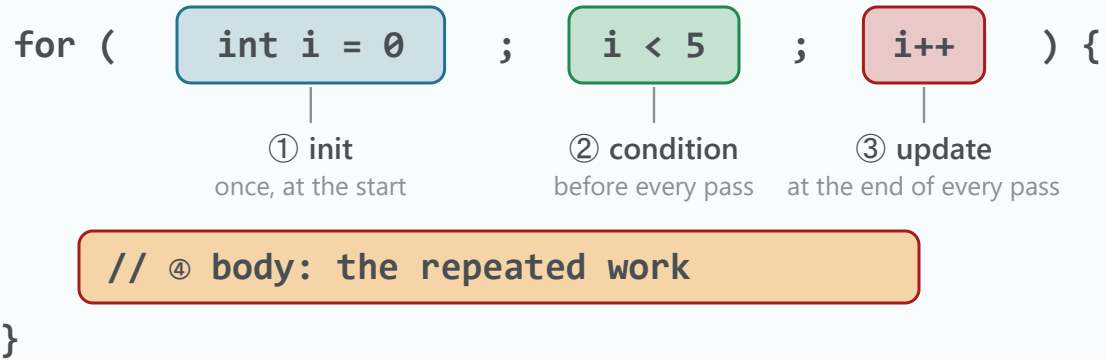
The condition comes last; the body runs **at least once**.

```
do { … } while (condition);
```



- Loops are handy for **processing a data set in order** and for repeating work; they matter especially with arrays and lists.
- One loop instead of writing the same line 100 times: the code becomes **shorter, tidier and more readable**.

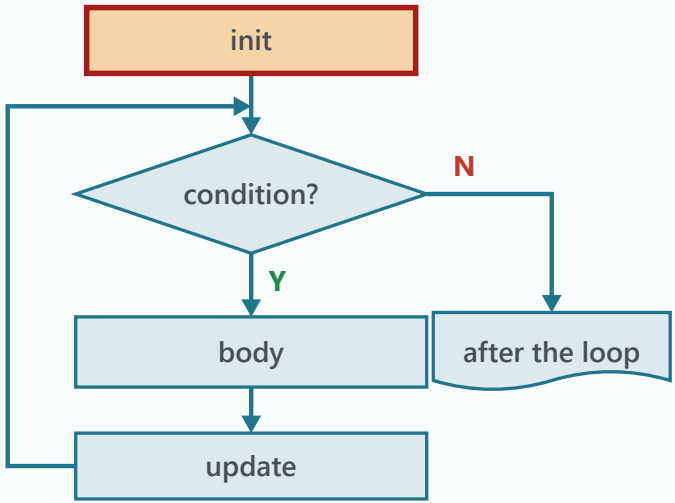
The for Loop



- The header sets the loop's **init**, **condition** and **update** (increment) steps on one line, separated by `;`.
- The condition is checked before every pass (iteration); the update step changes the loop variable.
- Order of execution: **1** → **2** → **4** → **3** → **2** → **4** → **3** → … → **2** false → exit the loop.
- `i` = 0, 1, 2, 3, 4: the body runs **5 times**; `i` reaching 5 makes the condition false. `i` exists only inside the loop.

How Loops Work

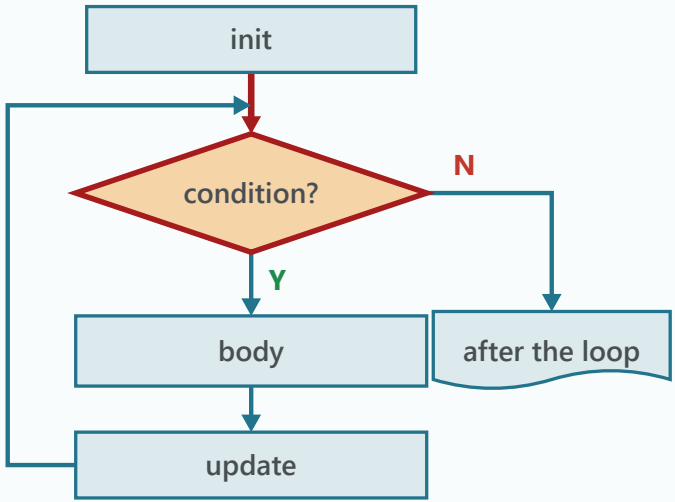
- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



1. Init: the loop variable gets its first value (once).

How Loops Work

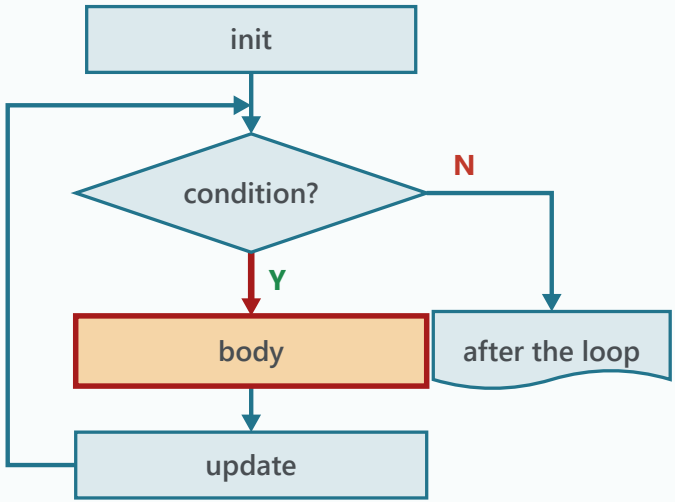
- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



2. The condition is checked.

How Loops Work

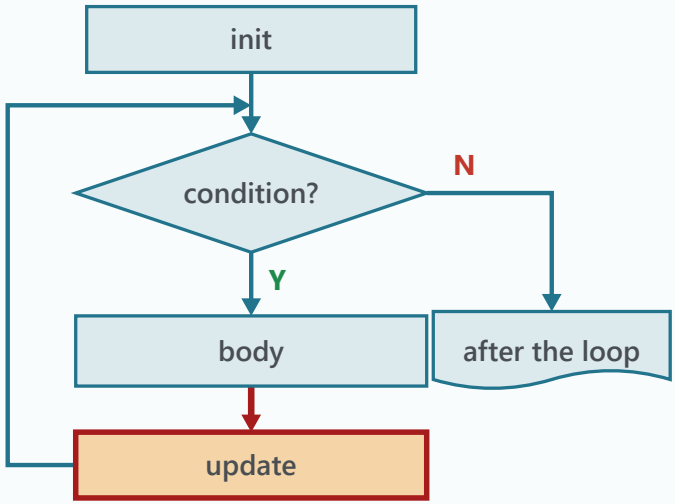
- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



3. If the condition is true, the body runs.

How Loops Work

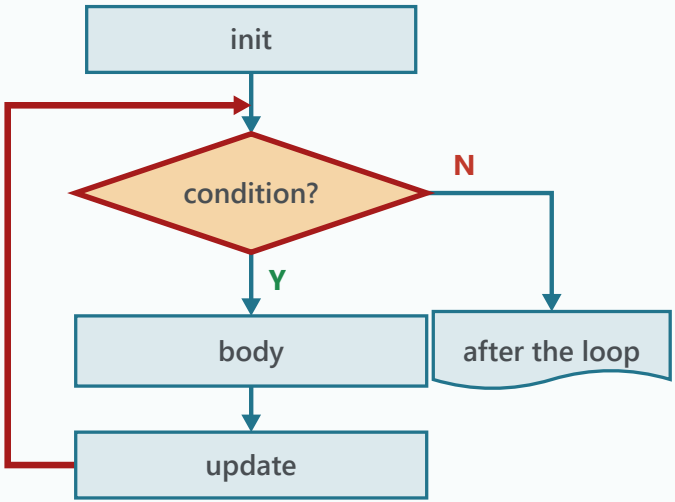
- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



4. Update: the loop variable changes.

How Loops Work

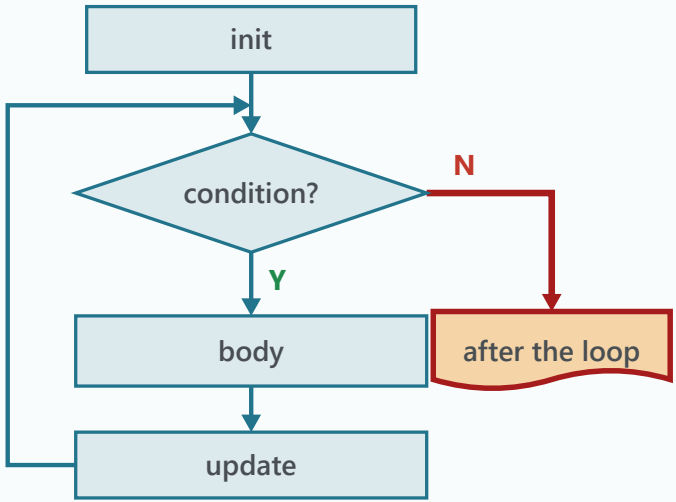
- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



5. Back to the condition; steps 3–4–5 repeat while it is true.

How Loops Work

- The loop **starts with the init step**.
- **The condition is checked.**
  - If it holds, the statements in the loop body run.
  - The update (increment) step is applied (`for` loops).
  - If the condition still holds, the body runs again.
- When the condition fails, **the loop exits** and execution continues with the next statement.
- If the condition is false on the first check, the body **never** runs.



6. When the condition is false, the loop exits.

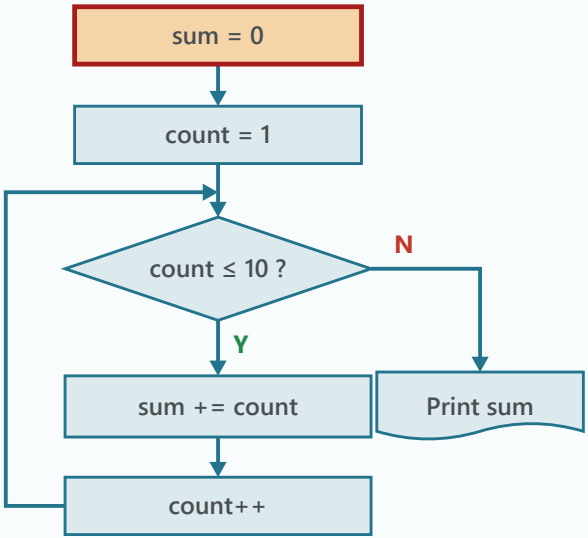
Summing Numbers (for)

```
▶ int sum = 0;
2 for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 0`

OUTPUT



`sum` starts at 0 (the identity element of addition).

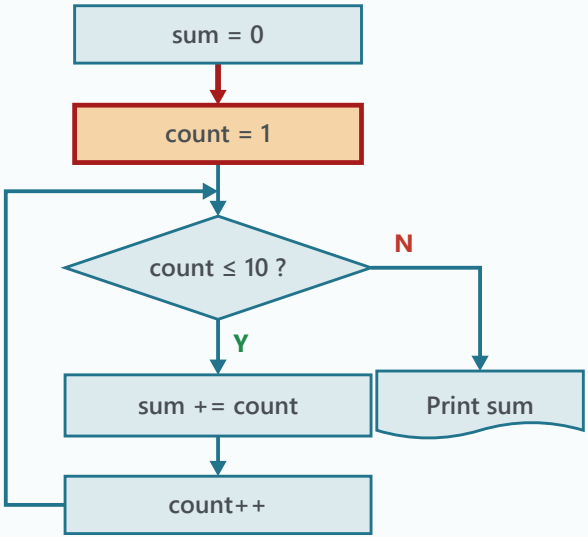
Summing Numbers (for)

```
1 int sum = 0;
▶ for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 0` `count = 1`

OUTPUT



Init: `count` = 1 (once).

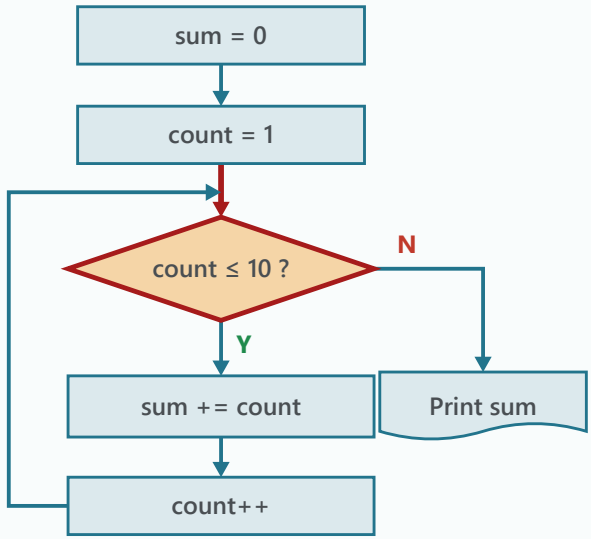
Summing Numbers (for)

```
1 int sum = 0;
▶ for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 0` `count = 1`

OUTPUT



Condition: 1 <= 10 → true.

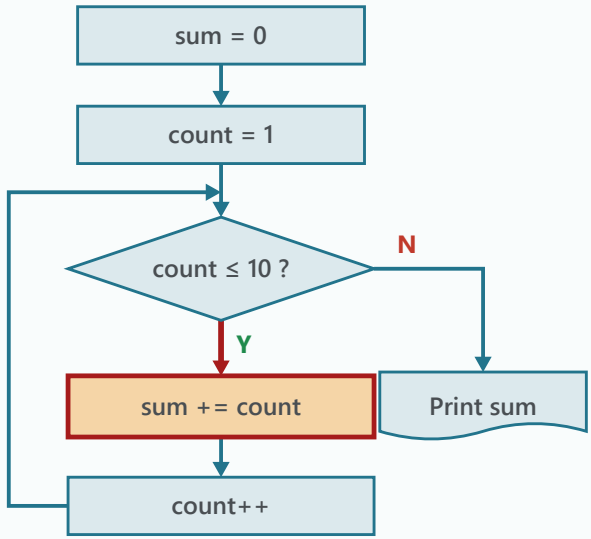
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 1` `count = 1`

OUTPUT



Pass 1: `sum` = 0 + 1 = 1.

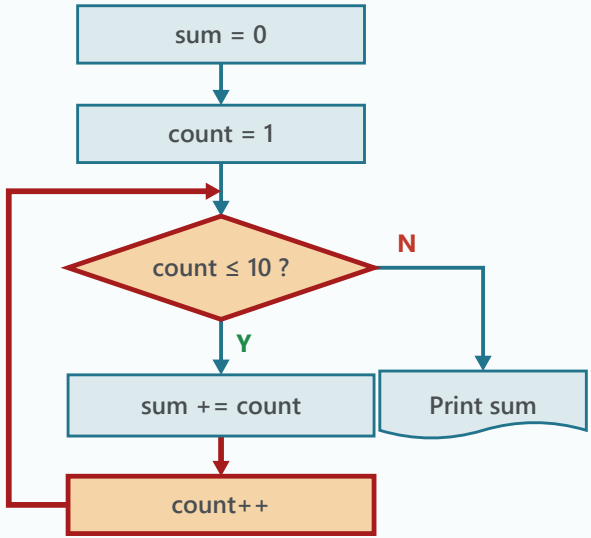
Summing Numbers (for)

```
1 int sum = 0;
▶ for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 1` `count = 2`

OUTPUT



`count++` → 2, condition: 2 <= 10 → true.

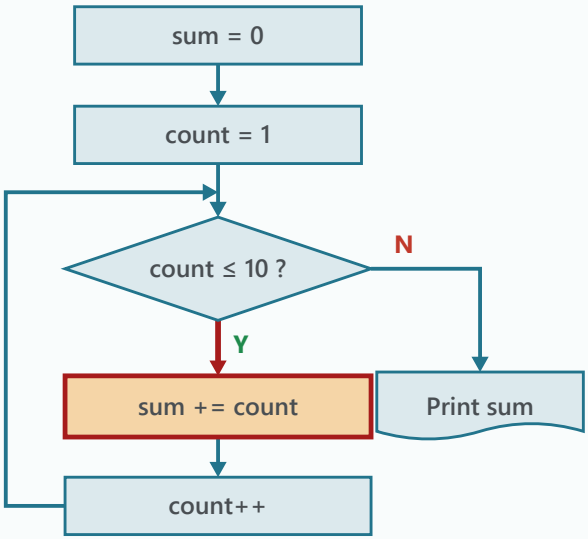
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 3` `count = 2`

OUTPUT



Pass 2: `sum` = 1 + 2 = 3.

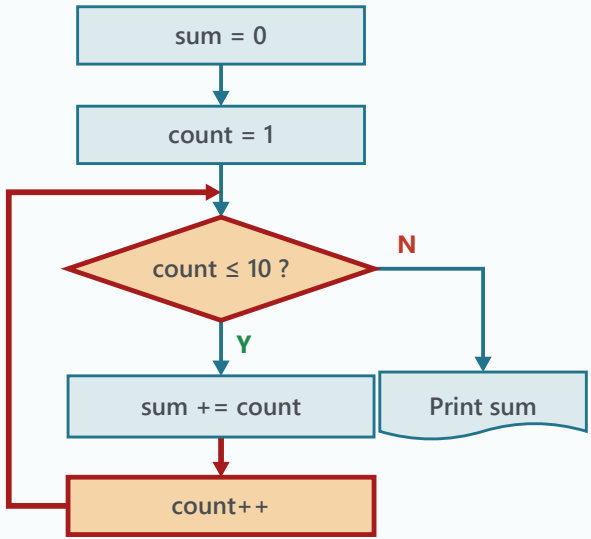
Summing Numbers (for)

```
1 int sum = 0;
▶ for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 3` `count = 3`

OUTPUT



`count++` → 3, condition: 3 <= 10 → true.

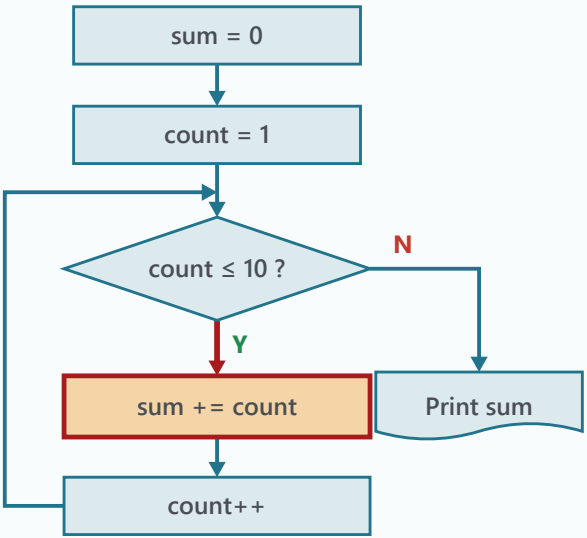
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 6` `count = 3`

OUTPUT



Pass 3: `sum` = 3 + 3 = 6. The update and condition repeat the same way on every pass.

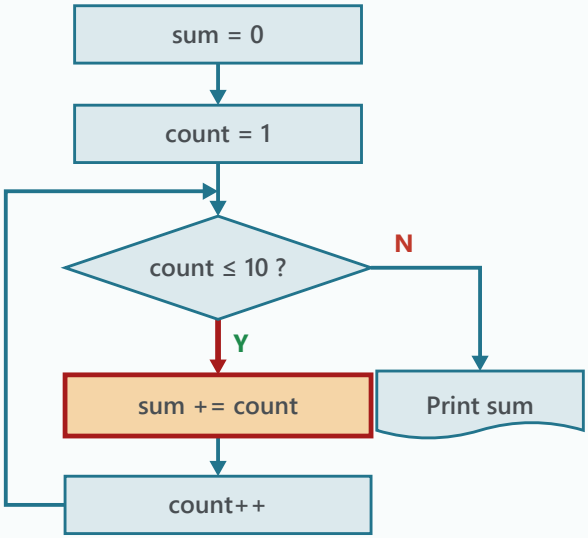
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 10` `count = 4`

OUTPUT



Pass 4: `sum` = 6 + 4 = 10.

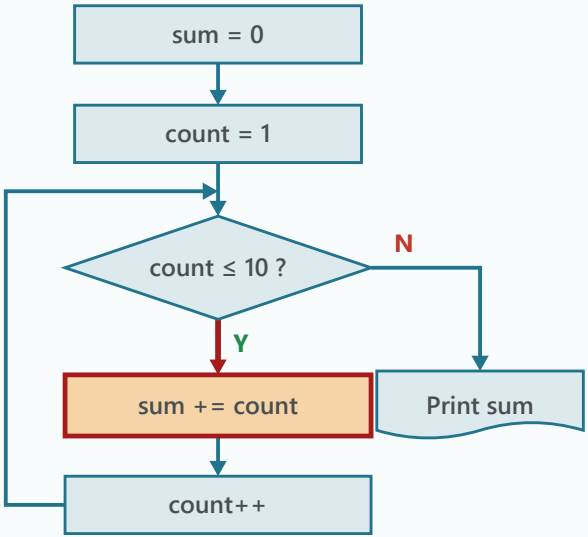
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 15` `count = 5`

OUTPUT



Pass 5: `sum` = 10 + 5 = 15.

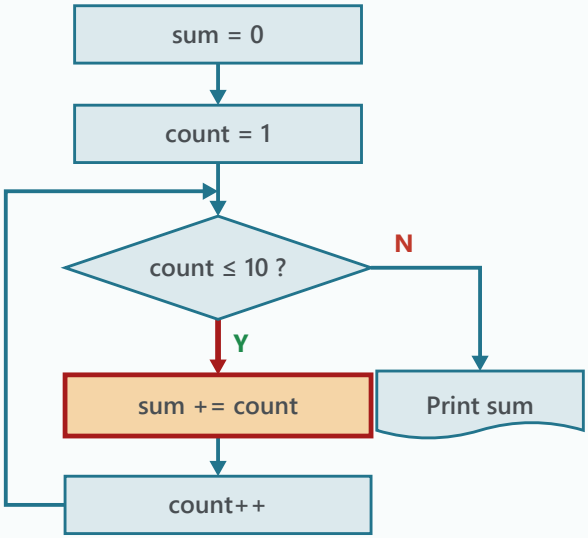
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 21` `count = 6`

OUTPUT



Pass 6: `sum` = 15 + 6 = 21.

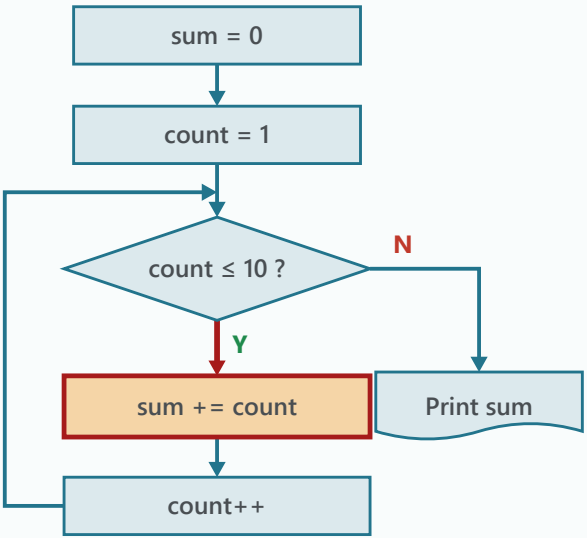
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 28` `count = 7`

OUTPUT



Pass 7: `sum` = 21 + 7 = 28.

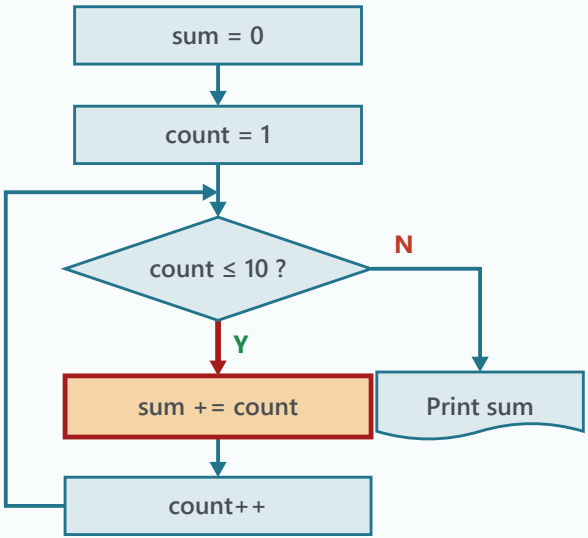
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 36` `count = 8`

OUTPUT



Pass 8: `sum` = 28 + 8 = 36.

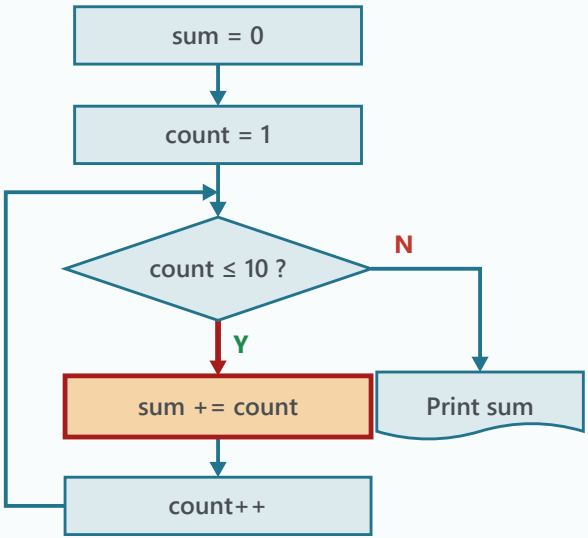
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 45` `count = 9`

OUTPUT



Pass 9: `sum` = 36 + 9 = 45.

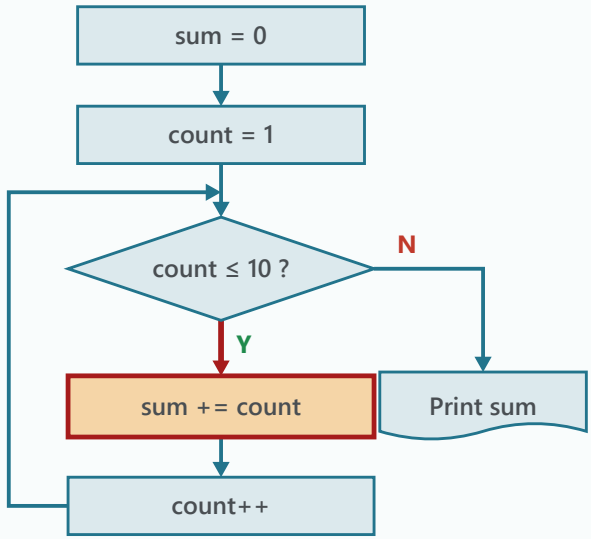
Summing Numbers (for)

```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
      sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 55` `count = 10`

OUTPUT



Pass 10: `sum` = 45 + 10 = 55.

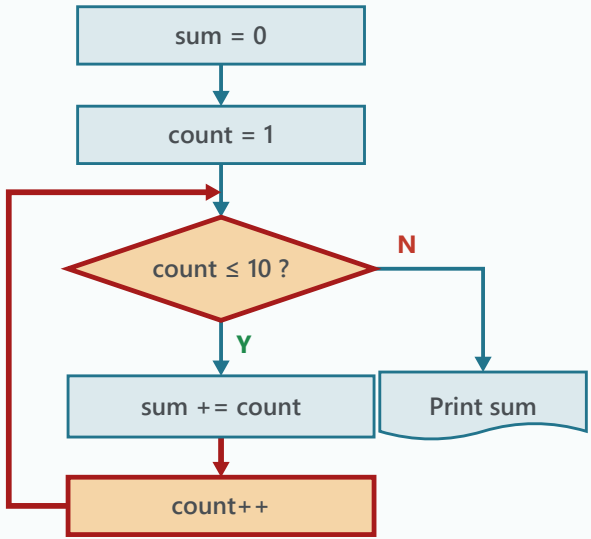
Summing Numbers (for)

```
1 int sum = 0;
▶ for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
5 System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 55` `count = 11`

OUTPUT



`count++` → 11, condition: 11 <= 10 → **false**. The loop ends; the body ran exactly 10 times.

Summing Numbers (for)

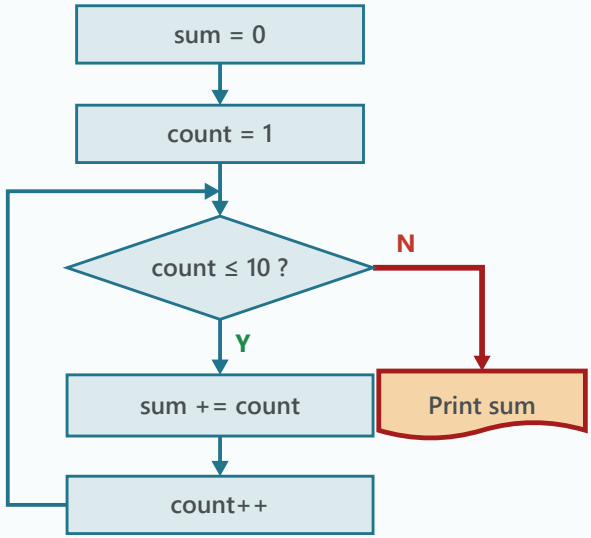
```
1 int sum = 0;
2 for (int count = 1; count <= 10; count++) {
3     sum += count;
4 }
▶ System.out.println("Sum: " + sum);
```

VARIABLES

`sum = 55`

OUTPUT

```
Sum: 55
```



After the loop, sum is printed. `count` no longer exists: it lives only inside the loop.

Summing a Range

```java
import java.util.Scanner;

public class RangeSum {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Start value (a): ");
        int a = input.nextInt();
        System.out.print("End value (b): ");
        int b = input.nextInt();
        int sum = 0;
        for (int i = a; i <= b; i++) {
            sum += i;
        }
        System.out.println("Sum of the numbers: " + sum);
    }
}
```

INPUTS: 3 AND 7

```
Start value (a): 3
End value (b): 7
Sum of the numbers: 25
```

The loop variable `i` starts not at a fixed number but at **the user's input** `a`: 3 + 4 + 5 + 6 + 7 = 25.

INPUTS: 7 AND 3

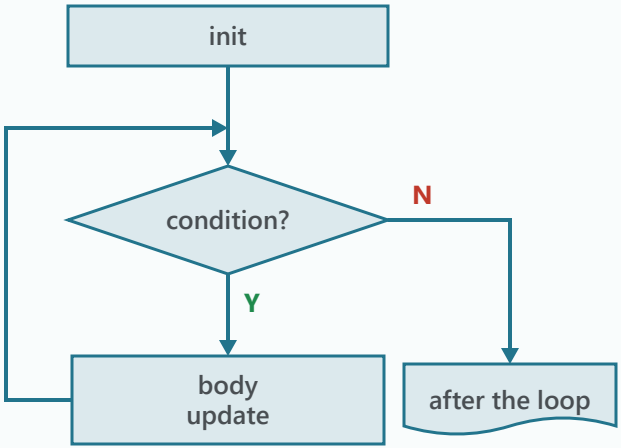
```
Start value (a): 7
End value (b): 3
Sum of the numbers: 0
```

On the first check 7 <= 3 is false: the body **never** runs, and the sum stays 0. Fix: if `a > b`, swap the two first (with an `if`).

The while Loop

```
int i = 0;
while (i < 5) {
    // statements in the loop
    i++;
}
```

- **Only a condition** is checked; init and update are written separately, outside and inside the loop.
- Runs **as long as the condition holds**; the condition is tested **before** the body.
- The natural choice when the number of repetitions isn't known in advance (e.g. “until the user enters 0”).
- **Beware of infinite loops!** `i++` missing → `i` stays 0, the condition stays true and the program never ends.



LOOPS

for vs. while

FOR

```
int sum = 0;
for (int count = 1; count <= 10; count++) {
    sum += count;
}
System.out.println("Sum: " + sum);
```

WHILE

```
int sum = 0;
int count = 1;
while (count <= 10) {
    sum += count;
    count++;
}
System.out.println("Sum: " + sum);
```

```
Sum: 55
```

Both loops have the same three parts: **init** `count = 1`, **condition** `count <= 10`, **update** `count++`. `for` puts them on one line; `while` scatters them, and the update is easy to forget. Both print 55.

Computing a Factorial (while)

```
▶ int n = 5;
▶ int factorial = 1;
▶ int count = 1;
4 while (count <= n) {
5     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

5! = 1 × 2 × 3 × 4 × 5

Goal: 5! = 1 × 2 × 3 × 4 × 5. The product starts at **1** (if it were 0, the result would always be 0).

Step 1 / 10

VARIABLES

`n = 5` `factorial = 1` `count = 1`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
▶     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

factorial = 1 × 1 = 1

1 <= 5 → true. `factorial *= count` : 1 × 1 = 1.

Step 2 / 10

VARIABLES

`n = 5` `factorial = 1` `count = 1`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
4 while (count <= n) {
5     factorial *= count;
▶     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

count = 2

`count++` → 2. Back to the condition.

Step 3 / 10

VARIABLES

`n = 5` `factorial = 1` `count = 2`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
▶     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

factorial = 1 × 2 = 2

2 <= 5 → true. `factorial *= count` : 1 × 2 = 2.

Step 4 / 10

VARIABLES

`n = 5` `factorial = 2` `count = 2`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
▶     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

factorial = 2 × 3 = 6

3 <= 5 → true. `factorial *= count` : 2 × 3 = 6.

Step 5 / 10

VARIABLES

`n = 5` `factorial = 6` `count = 3`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
▶     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

factorial = 6 × 4 = 24

4 <= 5 → true. `factorial *= count` : 6 × 4 = 24.

Step 6 / 10

VARIABLES

`n = 5` `factorial = 24` `count = 4`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
▶     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

factorial = 24 × 5 = 120

5 <= 5 → true. `factorial *= count` : 24 × 5 = 120.

Step 7 / 10

VARIABLES

`n = 5` `factorial = 120` `count = 5`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
4 while (count <= n) {
5     factorial *= count;
▶     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

count = 6

`count++` → 6. Back to the condition.

Step 8 / 10

VARIABLES

`n = 5` `factorial = 120` `count = 6`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
▶ while (count <= n) {
5     factorial *= count;
6     count++;
7 }
8 System.out.println(n + " factorial: " + factorial);
```

6 <= 5 → false

The condition is false: the loop exits.

Step 9 / 10

VARIABLES

`n = 5` `factorial = 120` `count = 6`

OUTPUT

Computing a Factorial (while)

```
1 int n = 5;
2 int factorial = 1;
3 int count = 1;
4 while (count <= n) {
5     factorial *= count;
6     count++;
7 }
▶ System.out.println(n + " factorial: " + factorial);
```

5! = 120

The result is printed. `int` is enough up to 12!; 13! overflows, so `long` is needed.

Step 10 / 10

VARIABLES

`n = 5` `factorial = 120` `count = 6`

OUTPUT

```
5 factorial: 120
```

Computing Powers (for)

```java
import java.util.Scanner;

public class Power {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter base and exponent: ");
        int base = sc.nextInt();
        int exp = sc.nextInt();
        long result = 1;
        for (int i = 1; i <= exp; i++) {
            result = result * base;
        }
        System.out.println("Result: " + result);
        System.out.println("Math.pow: " + Math.pow(base, exp));
        sc.close();
    }
}
```

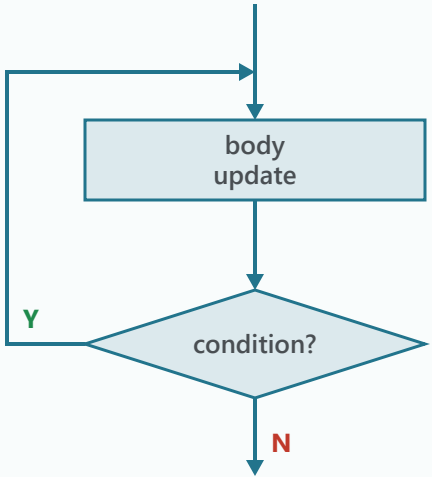
```
Enter base and exponent: 3 4
Result: 81
Math.pow: 81.0
```

Java's built-in `Math.pow(base, exp)` method does the same job but returns a `double` (81.0). Our own loop shows how the algorithm works; real programs prefer the library methods.

The do-while Loop

```
int i = 0;
do {
    // statements in the loop
    i++;
} while (i < 5);
```

- **Does the work first**, then checks the condition.
- Repeats while the condition holds.
- The body runs **at least once**.
- `while (condition)` must be followed by a **semicolon**.
- Typical uses: asking again until the user gives a valid value; menus.



do-while: Input Validation

```java
import java.util.Scanner;

public class DoWhileInput {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int num;
        do {
            System.out.print("Enter a number (1-10): ");
            num = input.nextInt();
        } while (num < 1 || num > 10);
        System.out.println("Accepted number: " + num);
    }
}
```

INPUTS: 15, 0, 7

```
Enter a number (1-10): 15
Enter a number (1-10): 0
Enter a number (1-10): 7
Accepted number: 7
```

- The question must be asked **at least once**: `do-while` is made exactly for this.
- 15 and 0 are out of range: the condition `num < 1 || num > 10` is true → ask again.
- 7 is valid: the condition is false → exit the loop.
- `num` is declared before the loop because it is used in the condition and after the loop.

Input Validation: Tracing the Code

```
Scanner input = new Scanner(System.in);
int num;
do {
    System.out.print("Enter a number (1-10): ");
    num = input.nextInt();
} while (num < 1 || num > 10);
System.out.println("Accepted number: " + num);
```

keyboard input

12 -3 4

The user will type **12**, **−3**, **4** in turn. `num` is declared before the loop; it is used in the condition and after the loop.

Step 1 / 8

VARIABLES

—

OUTPUT

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
▶     System.out.print("Enter a number (1-10): ");
▶     num = input.nextInt();
6 } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```

keyboard input

12 | -3 | 4

num = 12

The question is asked without checking the condition. `nextInt()` reads the first input: `num = 12`.

Step 2 / 8

VARIABLES

`num = 12`

OUTPUT

```
Enter a number (1-10): 12
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
4     System.out.print("Enter a number (1-10): ");
5     num = input.nextInt();
▶ } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```

keyboard input

12 -3 4

num = 12

12 < 1 || 12 > 10 → true

ask again

`12 < 1 || 12 > 10` : the right side is true → the condition is **true**, so the question is asked again.

Step 3 / 8

VARIABLES

num = **12**

OUTPUT

```
Enter a number (1-10): 12
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
▶     System.out.print("Enter a number (1-10): ");
▶     num = input.nextInt();
6 } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```

keyboard input



The second input is read: `num = −3`.

Step 4 / 8

VARIABLES

`num = −3`

OUTPUT

```
Enter a number (1-10): 12
Enter a number (1-10): -3
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
4     System.out.print("Enter a number (1-10): ");
5     num = input.nextInt();
▶ } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```

keyboard input

12 | -3 | 4

num = -3

-3 < 1 || -3 > 10 → true

ask again

`-3 < 1` is true; `||` skips the right side → the condition is **true**, again.

Step 5 / 8

VARIABLES

`num = -3`

OUTPUT

```
Enter a number (1-10): 12
Enter a number (1-10): -3
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
▶     System.out.print("Enter a number (1-10): ");
▶     num = input.nextInt();
6 } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```



Third input: `num = 4`.

Step 6 / 8

VARIABLES

`num = 4`

OUTPUT

```
Enter a number (1-10): 12
Enter a number (1-10): -3
Enter a number (1-10): 4
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
4     System.out.print("Enter a number (1-10): ");
5     num = input.nextInt();
▶ } while (num < 1 || num > 10);
7 System.out.println("Accepted number: " + num);
```

keyboard input

12 -3 **4**

`num = 4`

4 < 1 || 4 > 10 → false exit

`4 < 1` false, `4 > 10` false → the condition is **false**: the loop exits.

Step 7 / 8

VARIABLES

`num = 4`

OUTPUT

```
Enter a number (1-10): 12
Enter a number (1-10): -3
Enter a number (1-10): 4
```

Input Validation: Tracing the Code

```
1 Scanner input = new Scanner(System.in);
2 int num;
3 do {
4     System.out.print("Enter a number (1-10): ");
5     num = input.nextInt();
6 } while (num < 1 || num > 10);
▶ System.out.println("Accepted number: " + num);
```

keyboard input

12 -3 **4** num = 4

The valid number is accepted. The question was asked 3 times; the loop runs until a valid number arrives.

Step 8 / 8

VARIABLES

num = 4

OUTPUT

```
Enter a number (1-10): 12
Enter a number (1-10): -3
Enter a number (1-10): 4
Accepted number: 4
```

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
8      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

i = 10

`i` = 10.

Step 1 / 8

VARIABLES

`i = 10`

OUTPUT

while vs. do-while

```
1  int i = 10;
▶  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
8      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

`i < 5 → 10 < 5 → false`

`while` tests its condition **first**: false. The body (lines 3–4) **never** runs.

Step 2 / 8

VARIABLES

`i = 10`

OUTPUT

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
   int j = 10;
7  do {
8      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

j = 10

`j` = 10.

Step 3 / 8

VARIABLES

`i = 10` `j = 10`

OUTPUT

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
▶  do {
8      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

do: enter without checking

The `do` block is entered with no check.

Step 4 / 8

VARIABLES

`i = 10` `j = 10`

OUTPUT

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
▶      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

do-while: 10

The body runs once.

Step 5 / 8

VARIABLES

`i = 10` `j = 10`

OUTPUT

```
do-while: 10
```

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
8      System.out.println("do-while: " + j);
▶      j++;
10 } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

j = 11

`j++` → 11.

Step 6 / 8

VARIABLES

`i = 10` `j = 11`

OUTPUT

```
do-while: 10
```

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
8      System.out.println("do-while: " + j);
9      j++;
▶ } while (j < 5);
11 System.out.println("i = " + i + ", j = " +
   j);
```

j < 5 → 11 < 5 → false

The condition is **then** tested: false → exit.

Step 7 / 8

VARIABLES

`i = 10` `j = 11`

OUTPUT

```
do-while: 10
```

while vs. do-while

```
1  int i = 10;
2  while (i < 5) {
3      System.out.println("while: " + i);
4      i++;
5  }
6  int j = 10;
7  do {
8      System.out.println("do-while: " + j);
9      j++;
10 } while (j < 5);
▶  System.out.println("i = " + i + ", j = " +
   j);
```

while: 0 passes · do-while: 1 pass

Same condition, different result: `while` ran 0 times, `do-while` ran once.

Step 8 / 8

VARIABLES

`i = 10` `j = 11`

OUTPUT

```
do-while: 10
i = 10, j = 11
```

Comparing the Loop Types

	`for`	`while`	`do-while`
Where is the condition tested?	Before every pass	Before every pass	After every pass
Minimum passes?	0	0	1
Init / update	In the header, on one line	Outside the loop / in the body	Outside the loop / in the body
When?	Known count: “from 1 to n”, “every element of an array”	Until a condition changes: “until the number is 0”	The work must happen at least once: input validation, menus
Syntax	`for (init; cond; upd) { }`	`while (cond) { }`	`do { } while (cond);`

The three loops can be converted into one another; choose for readability.

LOOPS

The for Loop: Playground

init 1 condition i <= 10 update i++ run

```
for (int i = 1; i <= 10; i++) { sum += i; }
```

1 2 3 4 5 6 7 8 9 10

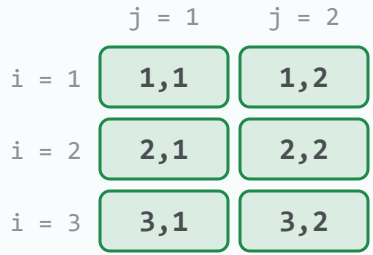
The body ran **10** times · sum = **55** · on exit i = **11** (11 <= 10 is false).

The boxes show the value of `i` on each pass the body ran. Try: init 10, condition `<=`, end 1 (never runs); update `i--` with `<=` (infinite loop); init 1, `!=` and `i += 2` with end 9 (stops) and 10 (never equal: infinite). The simulation stops after 200 passes.

Nested Loops

- **A nested loop** is a loop inside the body of another loop.
- Used for more complex work: **matrices** and multidimensional arrays, tables, **patterns**.
- Analogy: a clock. Each time the hour hand (outer loop) moves one step, the minute hand (inner loop) goes all the way round.
- The total number of passes is a product: 3 outer × 2 inner = **6** runs of the body.
- The number of nested loops and their conditions affect complexity. **Too much nesting** leads to complex, slow code: three loops × 1000 passes = 1 billion operations!

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        // nested loop body
    }
}
```



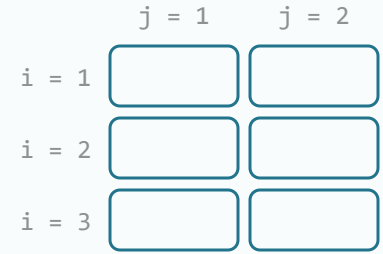
How Nested Loops Work

```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
3         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 1`

OUTPUT



Outer loop starts: `i` = 1, 1 <= 3 → true.

Step 1 / 10

How Nested Loops Work

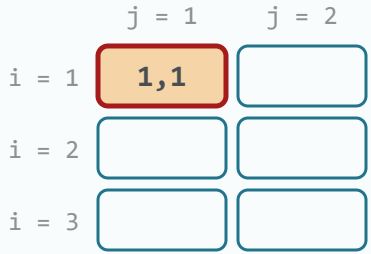
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 1` `j = 1`

OUTPUT

```
i = 1, j = 1
```



Inner loop starts: `j` = 1. Body: `i` = 1, `j` = 1 is printed.

Step 2 / 10

How Nested Loops Work

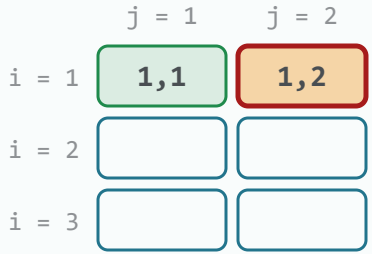
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 1` `j = 2`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
```



Body: `i` = 1, `j` = 2 is printed.

Step 3 / 10

How Nested Loops Work

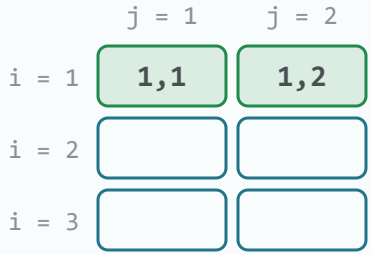
```
1 for (int i = 1; i <= 3; i++) {
▶       for (int j = 1; j <= 2; j++) {
3           System.out.println("i = " + i + ", j = " + j);
4       }
5 }
```

VARIABLES

`i = 1` `j = 3`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
```



`j` = 3 → 3 <= 2 false: the inner loop **is done**. Back to the outer loop: `i++` → 2, 2 <= 3 → true; the inner loop **starts over**.

How Nested Loops Work

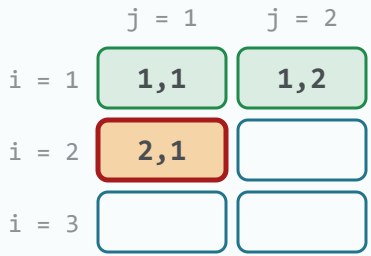
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 2` `j = 1`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
```



Inner loop starts: `j` = 1. Body: `i` = 2, `j` = 1 is printed.

Step 5 / 10

How Nested Loops Work

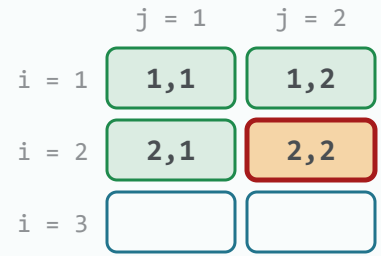
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 2` `j = 2`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
```



Body: `i` = 2, `j` = 2 is printed.

Step 6 / 10

How Nested Loops Work

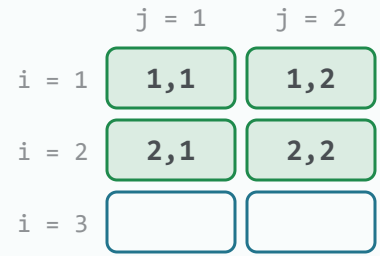
```
1 for (int i = 1; i <= 3; i++) {
▶       for (int j = 1; j <= 2; j++) {
3           System.out.println("i = " + i + ", j = " + j);
4       }
5 }
```

VARIABLES

`i = 2` `j = 3`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
```



`j` = 3 → 3 <= 2 false: the inner loop **is done**. Back to the outer loop: `i++` → 3, 3 <= 3 → true; the inner loop **starts over**.

How Nested Loops Work

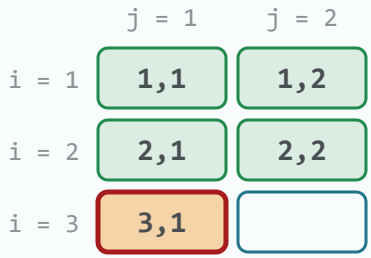
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 3` `j = 1`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
```



Inner loop starts: `j` = 1. Body: `i` = 3, `j` = 1 is printed.

How Nested Loops Work

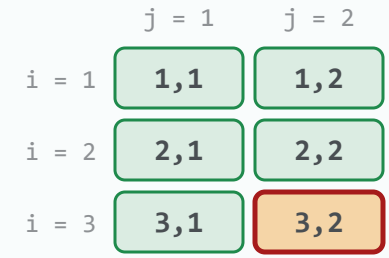
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
▶         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 3` `j = 2`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```



Body: `i` = 3, `j` = 2 is printed.

Step 9 / 10

How Nested Loops Work

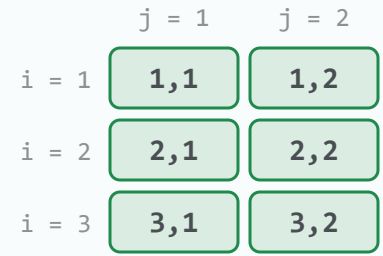
```
1 for (int i = 1; i <= 3; i++) {
2     for (int j = 1; j <= 2; j++) {
3         System.out.println("i = " + i + ", j = " + j);
4     }
5 }
```

VARIABLES

`i = 4`

OUTPUT

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```



`j` = 3 → 3 <= 2 false: the inner loop **is done**. `i++` → 4, 4 <= 3 false: the outer loop ends too. The body ran 3 × 2 = 6 times.

Step 10 / 10

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

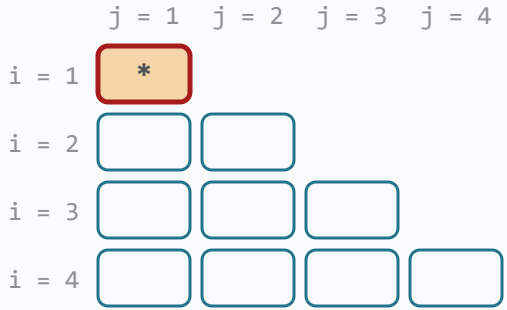
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
▶         System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 1` `j = 1`

OUTPUT

```
*
```



Row 1: the inner loop runs `j` from 1 to `i` = 1. `print("* ")` : stays on the same line.

Step 1 / 14

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

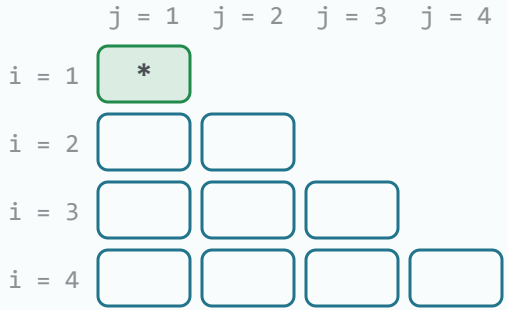
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
3         System.out.print("* ");
4     }
▶     System.out.println();
6 }
```

VARIABLES

`i = 1` `j = 2`

OUTPUT

```
*
```



Inner loop done (`j` = 2 > 1). `println()` moves to the next line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

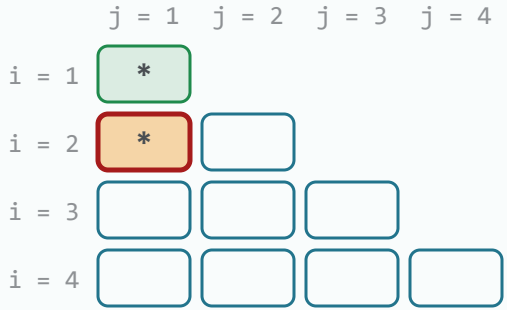
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
          System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 2` `j = 1`

OUTPUT

```
*
*
```



Row 2: the inner loop runs `j` from 1 to `i` = 2. `print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

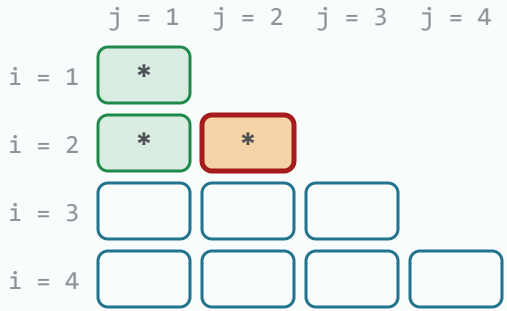
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
▶         System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 2` `j = 2`

OUTPUT

```
*
* *
```



`print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

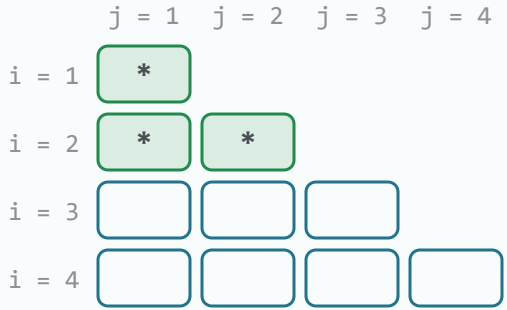
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
3         System.out.print("* ");
4     }
▶     System.out.println();
6 }
```

VARIABLES

`i = 2` `j = 3`

OUTPUT

```
*
* *
```



Inner loop done (`j` = 3 > 2). `println()` moves to the next line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

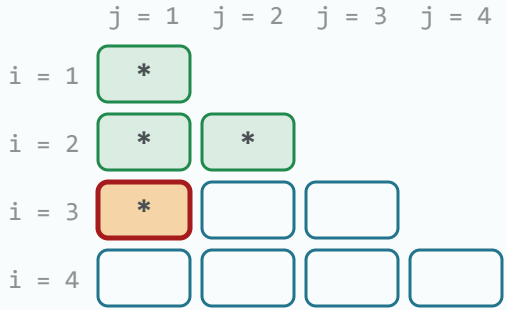
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
          System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 3` `j = 1`

OUTPUT

```
*
* *
*
```



Row 3: the inner loop runs `j` from 1 to `i` = 3. `print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

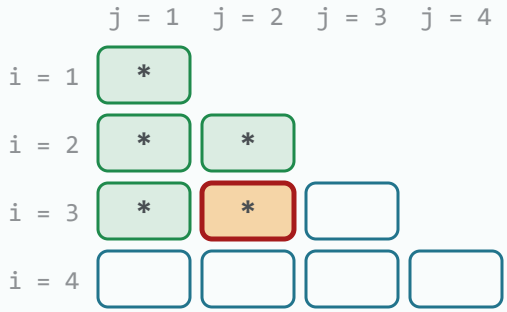
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
▶         System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 3` `j = 2`

OUTPUT

```
*
* *
* *
```



`print("* ")` : stays on the same line.

Step 7 / 14

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

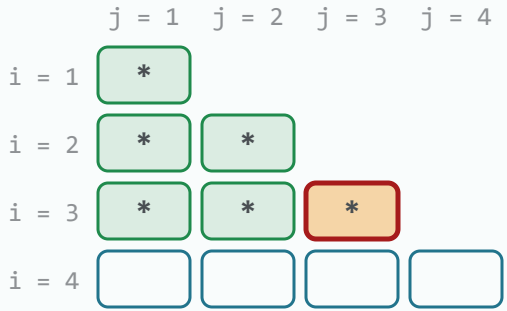
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
          System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 3` `j = 3`

OUTPUT

```
*
* *
* * *
```



`print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

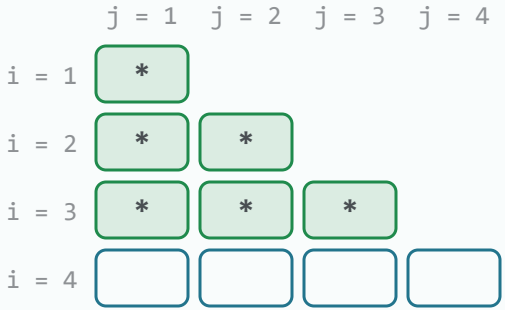
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
3         System.out.print("* ");
4     }
▶     System.out.println();
6 }
```

VARIABLES

`i = 3` `j = 4`

OUTPUT

```
*
* *
* * *
```



Inner loop done (`j` = 4 > 3). `println()` moves to the next line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

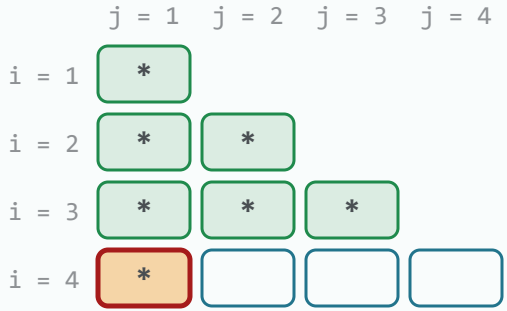
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
          System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 4` `j = 1`

OUTPUT

```
*
* *
* * *
*
```



Row 4: the inner loop runs `j` from 1 to `i` = 4. `print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

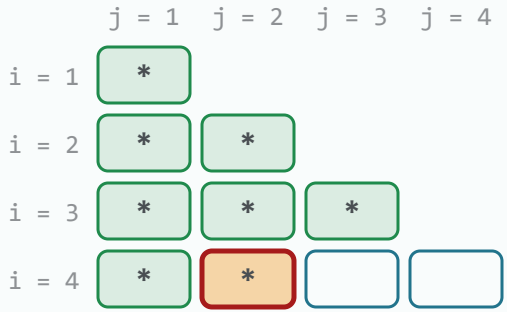
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
          System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 4` `j = 2`

OUTPUT

```
*
* *
* * *
* *
```



`print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

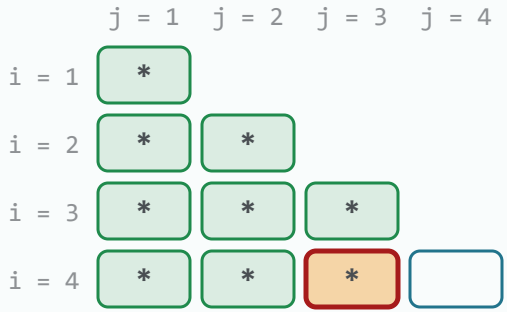
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
▶         System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 4` `j = 3`

OUTPUT

```
*
* *
* * *
* * *
```



`print("* ")` : stays on the same line.

Step 12 / 14

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

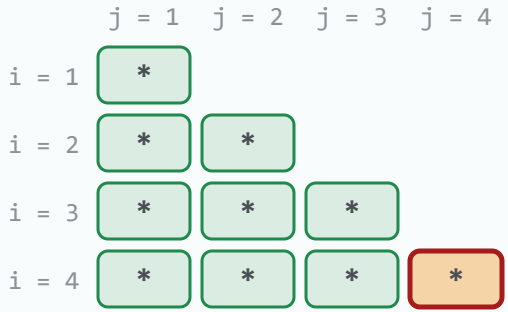
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
▶         System.out.print("* ");
4     }
5     System.out.println();
6 }
```

VARIABLES

`i = 4` `j = 4`

OUTPUT

```
*
* *
* * *
* * * *
```



`print("* ")` : stays on the same line.

Example: Star Pattern

The inner loop's bound depends on the outer loop's variable (`j <= i`): each row prints one more star.

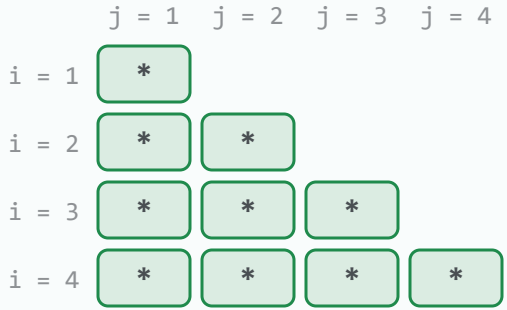
```
1 for (int i = 1; i <= 4; i++) {
2     for (int j = 1; j <= i; j++) {
3         System.out.print("* ");
4     }
▶     System.out.println();
6 }
```

VARIABLES

`i = 4` `j = 5`

OUTPUT

```
*
* *
* * *
* * * *
```



Inner loop done (`j` = 5 > 4). `println()` moves to the next line. 1 + 2 + 3 + 4 = 10 stars in total.

Place Value: What Is Printed?

```
for (int tens = 0; tens <= 4; tens++) {
    for (int ones = 1; ones <= 9; ones++) {
        int num = tens * 10 + ones;
        System.out.println("Tens digit: " + tens
                + ", ones digit: " + ones + " --> " + num);
    }
}
```

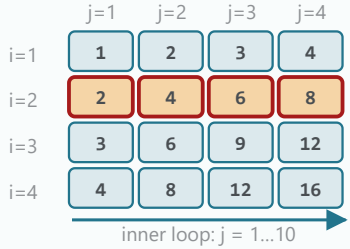
Question: how many lines does the program print? What are the first and last lines? Which numbers are **never** printed?

▶ **Answer**

Multiplication Table

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        int product = i * j;
        System.out.print(i + "x" + j + "=" + product + "\t");
    }
    System.out.println(); // new line
}
```

- The outer loop walks the **rows** (`i`), the inner one the **columns** (`j`): 10 × 10 = 100 products.
- `print` stays on the line; `"\t"` (tab) aligns columns.
- `println()` then starts a new row: **outside** the inner loop, **inside the outer one**.



EXCERPT OF THE OUTPUT (TABS SHOWN AS SPACES)

```
1x1=1   1x2=2   1x3=3   1x4=4   1x5=5   1x6=6   …
2x1=2   2x2=4   2x3=6   2x4=8   2x5=10   2x6=12   …
3x1=3   3x2=6   3x3=9   3x4=12   3x5=15   3x6=18   …
…
10x1=10   10x2=20   10x3=30   10x4=40   10x5=50   …
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break;
4     }
▶     System.out.println("Loop step: " + i);
6 }
7 System.out.println("Left the loop.");
```

i



`i` = 1: 1 == 5 false, `break` is skipped; the step is printed.

Step 1 / 6

VARIABLES

`i = 1`

OUTPUT

```
Loop step: 1
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break;
4     }
▶     System.out.println("Loop step: " + i);
6 }
7 System.out.println("Left the loop.");
```



`i` = 2: 2 == 5 false → printed.

Step 2 / 6

VARIABLES

`i = 2`

OUTPUT

```
Loop step: 1
Loop step: 2
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break;
4     }
▶     System.out.println("Loop step: " + i);
6 }
7 System.out.println("Left the loop.");
```



`i` = 3: 3 == 5 false → printed.

Step 3 / 6

VARIABLES

`i = 3`

OUTPUT

```
Loop step: 1
Loop step: 2
Loop step: 3
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break;
4     }
▶     System.out.println("Loop step: " + i);
6 }
7 System.out.println("Left the loop.");
```



`i` = 4: 4 == 5 false → printed.

Step 4 / 6

VARIABLES

`i = 4`

OUTPUT

```
Loop step: 1
Loop step: 2
Loop step: 3
Loop step: 4
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
▶       if (i == 5) {
▶           break;
4       }
5       System.out.println("Loop step: " + i);
6   }
7   System.out.println("Left the loop.");
```



`i` = 5: condition true → `break`. The loop ends **immediately**; 5 is not printed and 6–10 are never tried.

Step 5 / 6

VARIABLES

`i = 5`

OUTPUT

```
Loop step: 1
Loop step: 2
Loop step: 3
Loop step: 4
```

break in Loops

`break` is used to **end** a loop: once its condition holds, execution leaves the loop.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i == 5) {
3         break;
4     }
5     System.out.println("Loop step: " + i);
6 }
▶ System.out.println("Left the loop.");
```



Statements after the loop continue. `i` no longer exists.

Step 6 / 6

VARIABLES

—

OUTPUT

```
Loop step: 1
Loop step: 2
Loop step: 3
Loop step: 4
Left the loop.
```

Prime Check: A Flag and `break`

```java
import java.util.Scanner;

public class PrimeCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();
        boolean isPrime = true;            // flag
        if (num < 2) isPrime = false;     // 0 and 1 are not prime
        for (int divisor = 2; divisor <= num / 2; divisor++) {
            if (num % divisor == 0) {      // divides evenly
                isPrime = false;
                break;                     // break out of the loop
            }
        }
        if (isPrime) {
            System.out.println(num + " is prime.");
        } else {
            System.out.println(num + " is not prime.");
        }
        sc.close();
    }
}
```

```
Enter a number: 21
21 is not prime.

Enter a number: 97
97 is prime.
```

Faster: if a number has a divisor, one of them is at most √num. With the condition `divisor * divisor <= num`, only 8 divisors are tried for 97 instead of 47.

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
▶     System.out.println("Odd number: " + i);
6 }
```

i



`i` = 1 is odd → condition false, the number is printed.

Step 1 / 11

VARIABLES

`i = 1`

OUTPUT

```
Odd number: 1
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
▶       if (i % 2 == 0) {
▶           continue;
4       }
5       System.out.println("Odd number: " + i);
6   }
```

i



✓

`i` = 2 is even → `continue` : line 5 is skipped; control goes straight to `i++` .

Step 2 / 11

VARIABLES

`i = 2`

OUTPUT

```
Odd number: 1
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
▶     System.out.println("Odd number: " + i);
6 }
```



`i` = 3 is odd → condition false, the number is printed.

Step 3 / 11

VARIABLES

`i = 3`

OUTPUT

```
Odd number: 1
Odd number: 3
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
▶       if (i % 2 == 0) {
▶           continue;
4       }
5       System.out.println("Odd number: " + i);
6   }
```



`i` = 4 is even → `continue` : line 5 is skipped; control goes straight to `i++` .

Step 4 / 11

VARIABLES

`i = 4`

OUTPUT

```
Odd number: 1
Odd number: 3
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
▶     System.out.println("Odd number: " + i);
6 }
```



`i` = 5 is odd → condition false, the number is printed.

Step 5 / 11

VARIABLES

`i = 5`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
▶     if (i % 2 == 0) {
▶         continue;
4     }
5     System.out.println("Odd number: " + i);
6 }
```



`i` = 6 is even → `continue` : line 5 is skipped; control goes straight to `i++` .

Step 6 / 11

VARIABLES

`i = 6`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
▶     System.out.println("Odd number: " + i);
6 }
```



`i` = 7 is odd → condition false, the number is printed.

Step 7 / 11

VARIABLES

`i = 7`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
▶     if (i % 2 == 0) {
▶         continue;
4     }
5     System.out.println("Odd number: " + i);
6 }
```



`i` = 8 is even → `continue` : line 5 is skipped; control goes straight to `i++` .

Step 8 / 11

VARIABLES

`i = 8`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
▶     System.out.println("Odd number: " + i);
6 }
```



`i` = 9 is odd → condition false, the number is printed.

Step 9 / 11

VARIABLES

`i = 9`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
1 for (int i = 1; i <= 10; i++) {
▶     if (i % 2 == 0) {
▶         continue;
4     }
5     System.out.println("Odd number: " + i);
6 }
```

i

1	2	3	4	5	6	7	8	9	10
✓	skip	✓	skip	✓	skip	✓	skip	✓	

`i` = 10 is even → `continue` : line 5 is skipped; control goes straight to `i++` .

Step 10 / 11

VARIABLES

`i = 10`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

continue in Loops

`continue` skips the rest of the current pass and moves on **to the next pass**.

```
▶ for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue;
4     }
5     System.out.println("Odd number: " + i);
6 }
```



11 <= 10 false: the loop ends **normally**.

`continue` does not end the loop; it only skips that pass.

Step 11 / 11

VARIABLES

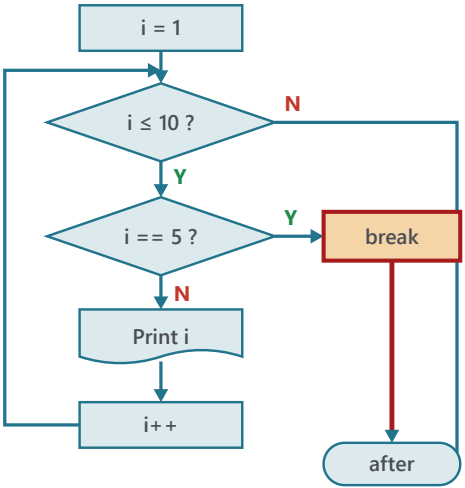
`i = 11`

OUTPUT

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

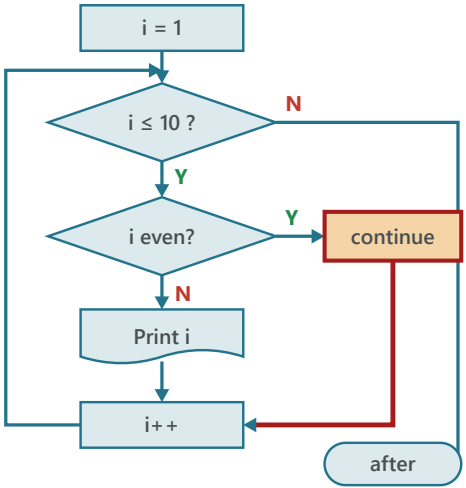
break vs. continue

break: ends the loop



The loop ends completely; execution moves to the statement after the loop.

continue: skips a pass



The rest of the pass is skipped; the loop goes on with the update and the condition.

Both are usually used inside an `if`; they also work in `while` and `do-while`; in `while`, a `continue` that skips the update can cause an infinite loop.

break in Nested Loops

BREAK ENDS ONLY THE INNER LOOP

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            break;
        }
        System.out.println(i + " " + j);
    }
}
```

```
1 1
2 1
3 1
```

LABELED BREAK: LEAVING THE OUTER LOOP

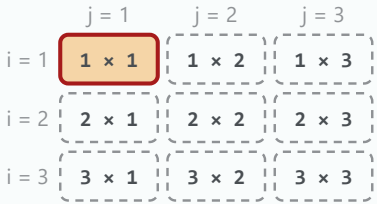
```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i * j == 4) {
            break outer;
        }
        System.out.println(i + " x " + j);
    }
}
```

```
1 x 1
1 x 2
1 x 3
2 x 1
```

In the left example, for each `i` only `j` = 1 is printed and the outer loop goes on. On the right, the `outer:` label names the outer loop; when `i * j == 4` (`i` = 2, `j` = 2) `break outer;` exits both loops at once. Use labels sparingly.

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
▶          if (i * j == 4) {
5              break outer;
6          }
▶          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



The `outer:` label names the outer loop. `i = 1, j = 1` : `1 × 1 = 1 ≠ 4` , printed.

Step 1 / 7

VARIABLES

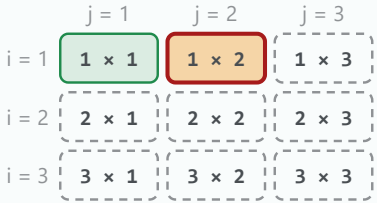
`i = 1` `j = 1`

OUTPUT

```
1 x 1
```

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
▶          if (i * j == 4) {
5              break outer;
6          }
▶          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



`j = 2` : `1 × 2 = 2` , printed.

Step 2 / 7

VARIABLES

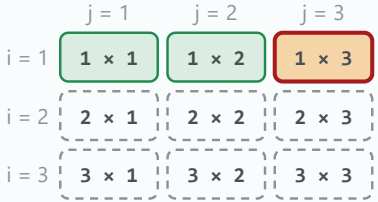
`i = 1` `j = 2`

OUTPUT

```
1 x 1
1 x 2
```

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
▶          if (i * j == 4) {
5              break outer;
6          }
▶          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



`j = 3`: `1 × 3 = 3`, printed. The inner loop ends; the outer loop moves to `i = 2`.

Step 3 / 7

VARIABLES

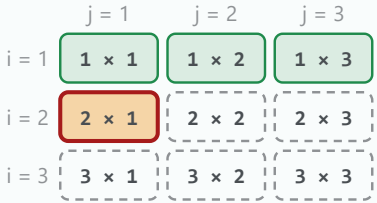
`i = 1` `j = 3`

OUTPUT

```
1 x 1
1 x 2
1 x 3
```

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
▶          if (i * j == 4) {
5              break outer;
6          }
▶          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



`i = 2, j = 1` : `2 × 1 = 2`, printed.

Step 4 / 7

VARIABLES

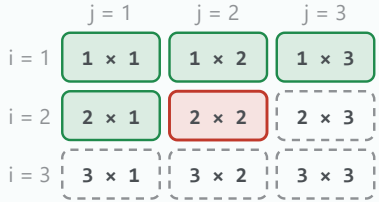
`i = 2` `j = 1`

OUTPUT

```
1 x 1
1 x 2
1 x 3
2 x 1
```

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
▶          if (i * j == 4) {
5              break outer;
6          }
7          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



`i = 2, j = 2` : `2 × 2 = 4` → the condition is true.

Step 5 / 7

VARIABLES

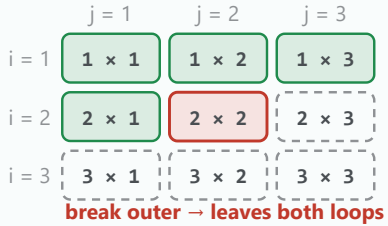
`i = 2` `j = 2`

OUTPUT

```
1 x 1
1 x 2
1 x 3
2 x 1
```

Labeled break: Tracing the Code

```
1  outer:
2  for (int i = 1; i <= 3; i++) {
3      for (int j = 1; j <= 3; j++) {
4          if (i * j == 4) {
▶              break outer;
6          }
7          System.out.println(i + " x " + j);
8      }
9  }
10 System.out.println("done");
```



`break outer;` exits not just the inner loop **but the labeled outer loop** too. The remaining cells are never visited.

Step 6 / 7

VARIABLES

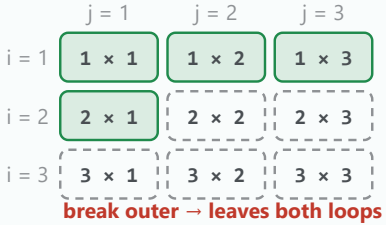
`i = 2` `j = 2`

OUTPUT

```
1 x 1
1 x 2
1 x 3
2 x 1
```

Labeled break: Tracing the Code

```
1 outer:
2 for (int i = 1; i <= 3; i++) {
3     for (int j = 1; j <= 3; j++) {
4         if (i * j == 4) {
5             break outer;
6         }
7         System.out.println(i + " x " + j);
8     }
9 }
▶ System.out.println("done");
```



Flow jumps past both loops: `done` is printed. With an unlabeled `break`, the `i = 3` pass would run too.

Step 7 / 7

VARIABLES

—

OUTPUT

```
1 x 1
1 x 2
1 x 3
2 x 1
done
```

Summary

Structure	What it does	Syntax
Sequential	Statements run top to bottom, once each	—
if / else	If the condition is true one block, otherwise the other	`if (c) { } else { }`
else if	The block of the first true condition runs	`if (c1) { } else if (c2) { } else { }`
switch	Compares a value with constant `case`s; exits with `break`	`switch (x) { case 1: … break; default: … }`
for	Init, condition and update on one line; counted repetition	`for (int i = 0; i < n; i++) { }`
while	While the condition is true; 0 or more passes	`while (c) { }`
do-while	Body first, then the condition; at least 1 pass	`do { } while (c);`
break / continue	Ends the loop / skips the pass	`break;` · `continue;`

SUMMARY

Common Mistakes: Decisions

Comparing with =

`if (x = 5)` → `error: incompatible types: int cannot be converted to boolean`. Correct: `x == 5`.

Semicolon after if

`if (x > 5); { … }` compiles, but the `;` is an empty statement: the block **always** runs.

Blockless body

`if (c) a(); b(); else c();` → `error: 'else' without 'if'`. Braces `{ }` are required for more than one statement.

Forgetting break

`switch`: with no `break`, the `case`s below run too (fall-through). The compiler doesn't warn you.

Assigning without default

If the variable is assigned only in the `case`s: `error: variable season might not have been initialized`. Fix: add a `default`.

Dot in decimal input

With a Turkish locale, `nextDouble()` given `-7.5` → `InputMismatchException`. `-7,5` is what you must type.

SUMMARY

Common Mistakes: Loops

Infinite loop

`while`: forgetting the update (`i++`) or updating in the wrong direction (`i--` with `i <= 10`). The compiler won't catch it; the program never stops.

Off-by-one error

For "from 1 to 10", writing `i < 10` skips 10; `i = 0` with `i <= 10` runs 11 times. Check the first and last pass separately.

Code after break

`break; System.out.println(i);` → `error: unreachable statement`. `break`: the line after it can never be reached.

; in the loop header

`for (int i = 0; i < 3; i++);` or `while (i < 5);` is a loop with an empty body; the block below doesn't belong to it.

Using the loop variable outside

`for (int i …) { }` followed by `println(i)` → `error: cannot find symbol`. If needed, declare `i` before the loop.

Sum/product initial value

A sum starts at 0, a product at **1**. `factorial = 0` makes the result always 0. Resetting the variable **inside** the loop is wrong too.

SUMMARY

Practice Problems (1): What Is Printed?

PROBLEM 1

```
int x = 7;
if (x > 5)
    if (x > 10)
        System.out.println("A");
    else
        System.out.println("B");
```

PROBLEM 2

```
int s = 0;
for (int i = 1; i <= 10; i += 3) {
    s += i;
}
System.out.println(s);
```

PROBLEM 3

```
int k = 2;
int points = 0;
switch (k) {
    case 1:  points += 10;
    case 2:  points += 20;
    case 3:  points += 30;  break;
    case 4:  points += 40;
}
System.out.println(points);
```

▶ **Answers**

SUMMARY

Practice Problems (2)

Problem 4

Write code that sums the numbers from 1 to 100 that are divisible by **3 or 5**.

Problem 5

Find how many **digits** an integer (e.g. 2024) has, using a `while` loop (`num /= 10` drops the last digit).

▶ **Answers**

Exercises (1)

Exercise 1

```
int n = 1;
while (n < 100) {
    n *= 3;
}
System.out.println(n);
```

Exercise 2

```
int k = 5;
do {
    k -= 2;
} while (k > 0);
System.out.println(k);
```

Exercise 3

```
int count = 0;
for (int i = 1; i <= 4; i++) {
    for (int j = i; j <= 4; j++) {
        count++;
    }
}
System.out.println(count);
```

▶ **Answers (solve on paper first, then run the code to check)**

Exercises (2)

Exercise 4

```
int t = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) {
        continue;
    }
    t += i;
}
System.out.println(t);
```

Exercise 5

```
int i = 10;
while (i >= 1) {
    System.out.print(i + " ");
    i -= 2;
}
System.out.println();
```

Exercise 6

```
int a = 17, b = 42;
int max = (a > b) ? a : b;
System.out.println(max);
```

▶ **Answers**

References

- Oracle, *The Java Tutorials*: “Control Flow Statements” (if-then-else, switch, while, do-while, for, branching statements).
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (Chapter 14: Blocks, Statements, and Patterns).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapters 4–5: Control Statements).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly (Chapter 5: Conditionals and Logic, Chapter 6: Loops).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed. (3.8: Control Flow).

The End

Chapter 3.1: Control Structures

Previous: Chapter 2.2: Syntax Rules · Next: Chapter 3.2: Methods