Chapter 1.4: Flowcharts

Object Oriented Programming with Java

Algorithms and flowcharts · Basic symbols · Drawing rules · Sequence · Selection · Iteration · Loop symbol · Subprograms · From flowchart to Java

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_01_04_Flowcharts.html

INTRODUCTION

Learning Outcomes

Concept

Explain how algorithms and flowcharts relate; say why and where flowcharts are used.

Symbols

Recognize and correctly use the Start/End, Input/Output, Process, Decision, Loop, Print, connector and subprogram symbols.

Control structures

Tell sequence, selection and iteration apart; draw a flowchart for each.

Tracing

Trace a flowchart step by step on a given input to find the output; spot boundary errors and infinite loops.

From flowchart to Java

Know the Java equivalent of each symbol: `nextInt()`, assignment, `if/else`, `while`, `for`, `println`.

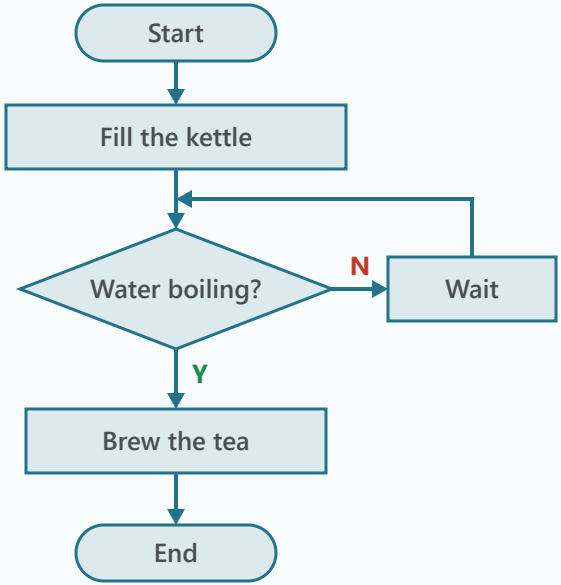
Design

Plan a simple problem (a sum, odd/even, counting) as a flowchart first, then turn it into a program.

Flowcharts

- **Algorithm:** a plan of step-by-step instructions for solving a problem or carrying out a task.
- **Flowchart:** shows an algorithm's steps, the links between them and its control logic **visually**.
- Each step is a **symbol**; their order is shown by **arrows** (flow lines).

Analogy: a recipe is an algorithm; the same recipe drawn with boxes and arrows is a flowchart. It is the easiest way to plan a solution on paper before writing a program.



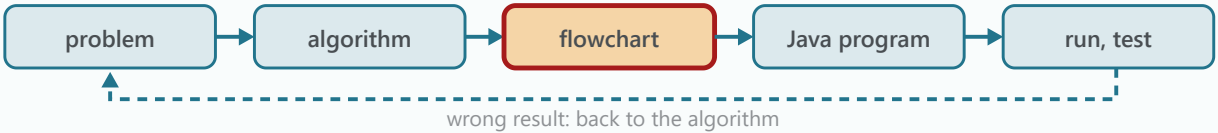
Benefits and Limitations

Benefits

+ **Simplifies** complex processes: the whole solution at a glance.
+ Helps find **errors** and room for improvement before any code is written.
+ Used to **document** and report processes.
+ **Language-independent**: a common language for the team.

Limitations

– For large programs the chart grows huge and hard to read.
– When the algorithm changes, the chart must be redrawn.
– More detail means more symbols; then **pseudocode** is more practical.



The usual workflow: **problem** → **algorithm** (steps) → **flowchart** → **Java program** → run and test.

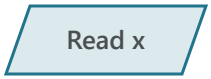
SYMBOLS AND RULES

The Basic Symbols



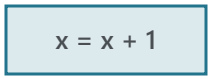
Start / End

Oval. Where the algorithm begins and ends.



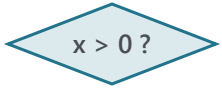
Input / Output

Parallelogram. Reading data from the keyboard (input).



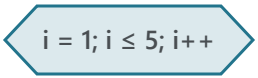
Process

Rectangle. Calculation, operation, assignment.



Decision

Diamond. A yes/no question with two exits.



Loop

Hexagon. A fixed number of repetitions.



Print

Wavy bottom edge. Writing the result to the screen or paper.



Connector

Small circle. Where lines join.



Flow line

Arrow. The order of steps; follow its direction.

The symbols follow ISO 5807. Some sources use a parallelogram for both input and output; the "Print" symbol is also common for screen output.

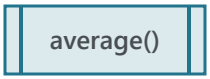
SYMBOLS AND RULES

The Java Equivalent of Each Symbol

Symbol	Text in the chart	Java
 Start / End	Start … End	`public static void main(String[] args) { … }`
 Input	Read num	`int num = scanner.nextInt();`
 Process	result = num1 + num2	`int result = num1 + num2;`
 Decision	result = 0 ?	`if (result == 0) { … } else { … }`
 Decision + back arrow	count ≤ 10 ?	`while (count <= 10) { … }`
 Loop	i = 1; i ≤ 10; i++	`for (int i = 1; i <= 10; i++) { … }`
 Print	Print result	`System.out.println(result);`

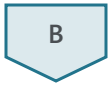
In a process box `=` means **assignment**; in a decision box it means **comparison**. Java writes assignment as `=` and comparison as `==`. The chart's "mod" is Java's `%` operator.

Other Symbols



Subprogram

Rectangle with double side lines. Calls a sequence of steps drawn separately. Java: a method call.



Off-page connector

Downward-pointing pentagon. The chart continues on another page; both ends carry the same letter.



Data store

Cylinder. Saving to and reading from a file or database.



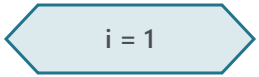
Manual input

Quadrilateral with a slanted top. Some sources draw keyboard input with it.



Annotation

Open bracket. A note outside the flow. Java: a `//` comment.



Preparation

Hexagon. ISO 5807 calls it preparation; used to set up a loop.

The basic symbols suffice for most charts. The extra ones help split long charts (subprogram, off-page connector) and show where data is stored.

How to Draw a Flowchart

Steps

1. Identify the operations step by step: inputs, calculations, outputs.
2. Place the steps in a logical order.
3. Connect all operations with **flow lines**.
4. Mark the **decision points** with conditions.

Drawing rules

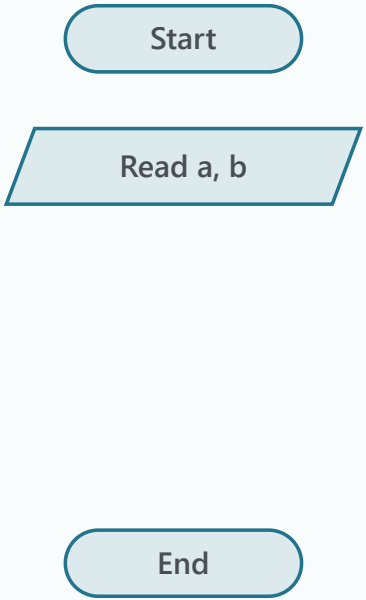
- Exactly one **Start** and at least one **End**.
- The flow reads top to bottom, left to right.
- Every symbol has one exit; only the **Decision** symbol has two (Y/N).
- Decision exits are always labeled **Yes/No**.
- Lines should not cross; use a connector if needed.
- Write short, clear text in the boxes.

Drawing Step by Step: Area of a Rectangle



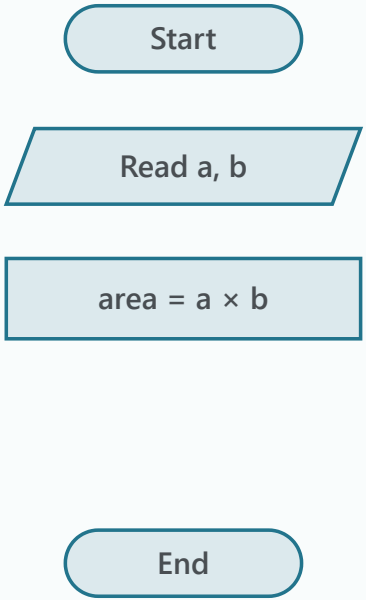
Problem: find **the area of a rectangle**. First the skeleton: a **Start** and an **End**.

Drawing Step by Step: Area of a Rectangle



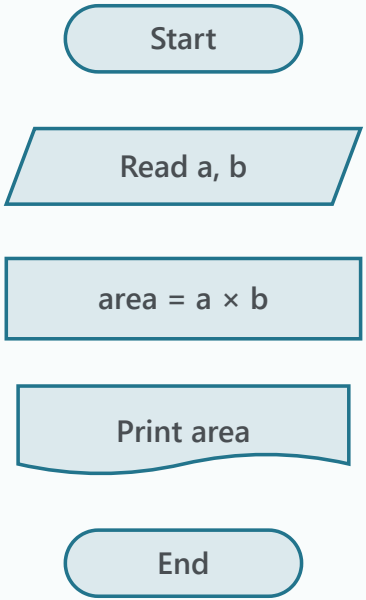
The **inputs**: side lengths a and b are read from the keyboard (parallelogram).

Drawing Step by Step: Area of a Rectangle



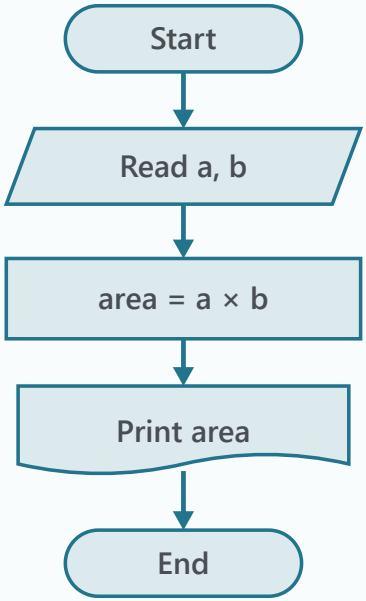
Add the **process**: compute `area = a × b` and assign it to the area variable (rectangle).

Drawing Step by Step: Area of a Rectangle



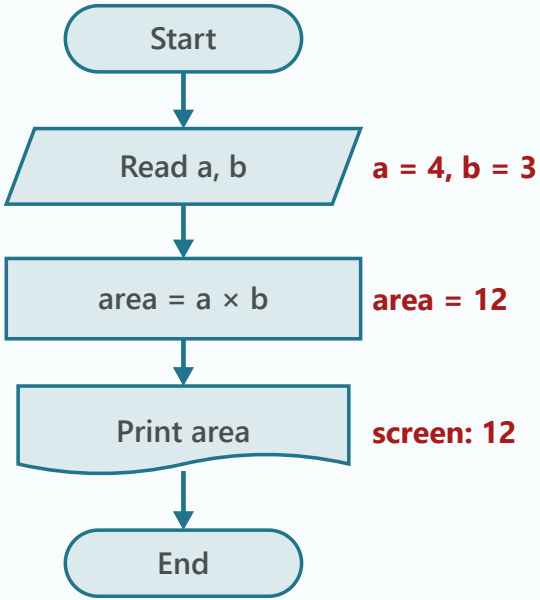
Add the **output**: print the computed area (Print).

Drawing Step by Step: Area of a Rectangle



Connect the symbols in order with **flow lines**. There is no decision here: the flow is **sequential**.

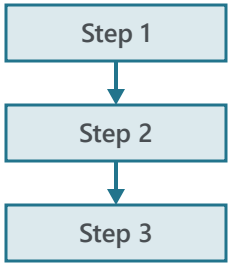
Drawing Step by Step: Area of a Rectangle



Test: with a = 4, b = 3 the area is 12 and the screen shows **12**. The chart works.

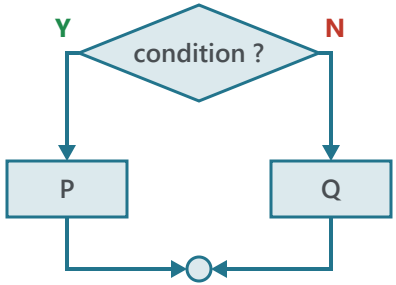
Control Structures

Sequence



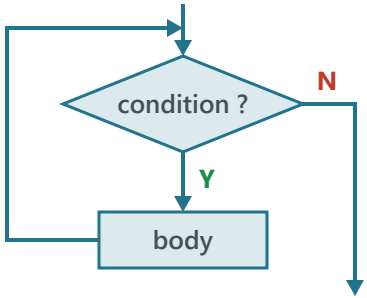
Steps run in order, once each.

Selection



The condition decides which steps run.

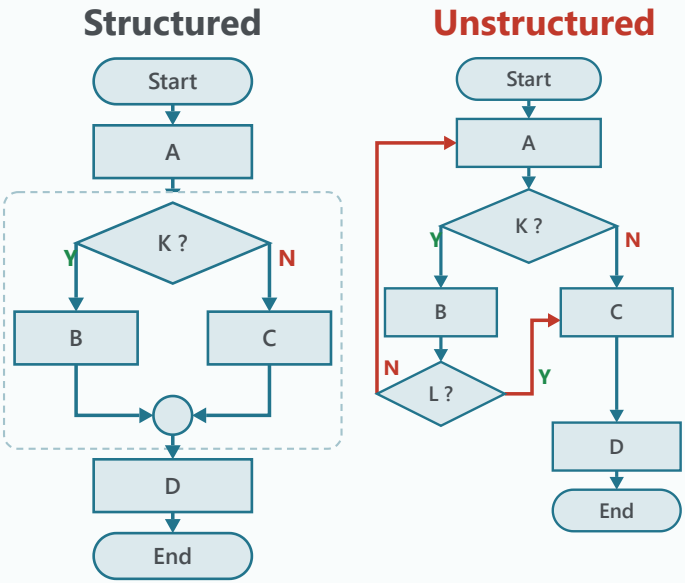
Iteration



Some steps repeat while a condition holds.

Structured program theorem (Böhm and Jacopini, 1966): every algorithm can be written using only these three structures (sequence, selection, iteration), nested. In Java: consecutive statements, `if/else`, `while` / `for`.

Structured and Unstructured Flowcharts

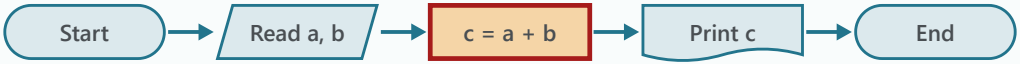


- In a **structured** chart every control structure has **one entry and one exit** (dashed frame). Structures are placed one after another or nested.
- In an **unstructured** chart arrows jump anywhere: L's Yes arrow enters the middle of the other branch (C), its No arrow goes back to A. Such “spaghetti” flow is hard to trace and debug.
- Older languages wrote such jumps with `goto`. In Java `goto` is a reserved word that cannot be used.
- `break` and `continue` are limited jumps: they only end the enclosing loop or skip to its next round.
- By the Böhm–Jacopini theorem every chart can be made structured, using a flag variable if necessary.

SEQUENCE

Sequential Flow

The steps (statements) are laid out one after another in a logical order. The flow runs top to bottom.



Simple
A plain, easy-to-follow flow with no branching or repetition.

Easy to maintain
Easy to debug and understand: the steps are traced in the order written.

Order matters
A step that uses an earlier result cannot move before it. Independent steps (e.g. two separate reads) can be swapped.

No going back
Each step runs exactly **once**; no step is revisited or skipped.

Wrong order: if `result = num1 + num2` comes before the numbers are read, the sum uses values that are not yet known. In Java the compiler catches variables declared but never assigned: `variable num1 might not have been initialized`.

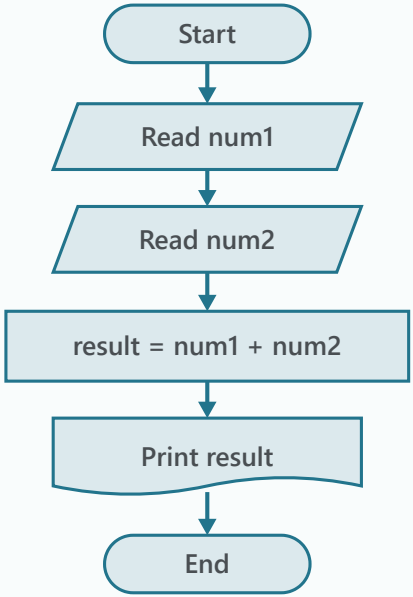
Example: The Sum of Two Numbers

Problem: print the sum of two numbers entered at the keyboard.

Algorithm

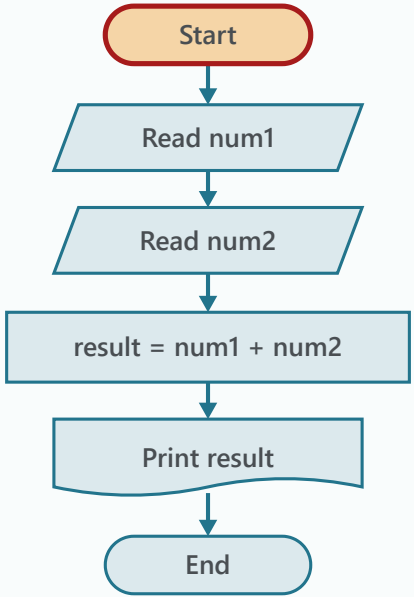
1. Start
2. Read the first number (num1)
3. Read the second number (num2)
4. Add them and assign to result
5. Print result
6. End

Each algorithm step corresponds to one symbol in the chart.



From Flowchart to Java: The Sum Program

```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          System.out.print("First number: ");
7          int num1 = scanner.nextInt();
8          System.out.print("Second number: ");
9          int num2 = scanner.nextInt();
10         int result = num1 + num2;
11         System.out.println("Result: " + result);
12         scanner.close();
13     }
14 }
```



VARIABLES

`num1 = –` `num2 = –` `result = –`

OUTPUT

The program starts at `main` (**Start**). A `Scanner` is created to read the keyboard.

From Flowchart to Java: The Sum Program

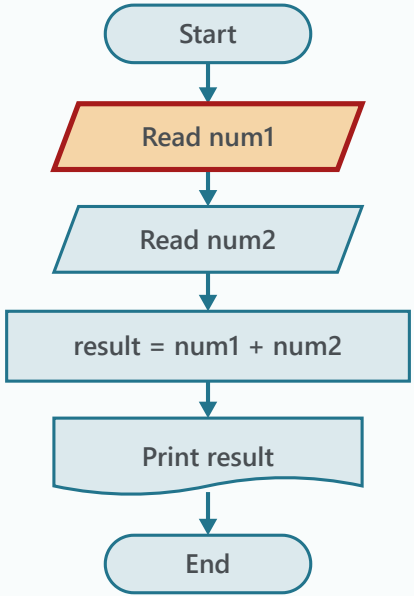
```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
           System.out.print("First number: ");
           int num1 = scanner.nextInt();
8          System.out.print("Second number: ");
9          int num2 = scanner.nextInt();
10         int result = num1 + num2;
11         System.out.println("Result: " + result);
12         scanner.close();
13     }
14 }
```

VARIABLES

`num1 = 7` `num2 = –` `result = –`

OUTPUT

```
First number: 7
```



The prompt prints and the user enters **7**: `nextInt()` assigns it to `num1`.

From Flowchart to Java: The Sum Program

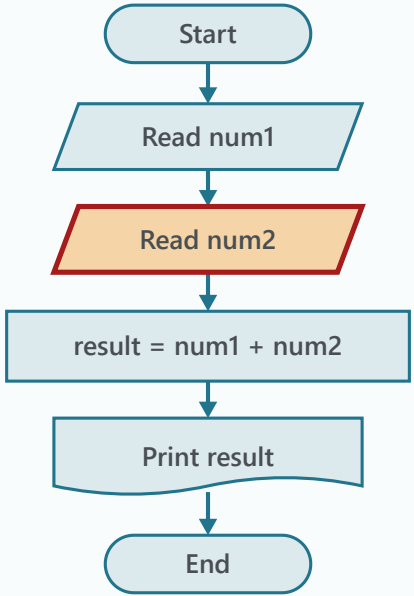
```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          System.out.print("First number: ");
7          int num1 = scanner.nextInt();
▶          System.out.print("Second number: ");
▶          int num2 = scanner.nextInt();
10         int result = num1 + num2;
11         System.out.println("Result: " + result);
12         scanner.close();
13     }
14 }
```

VARIABLES

`num1 = 7` `num2 = 5` `result = –`

OUTPUT

```
First number: 7
Second number: 5
```



The user enters **5**: `num2 = 5`.

From Flowchart to Java: The Sum Program

```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          System.out.print("First number: ");
7          int num1 = scanner.nextInt();
8          System.out.print("Second number: ");
9          int num2 = scanner.nextInt();
▶          int result = num1 + num2;
11         System.out.println("Result: " + result);
12         scanner.close();
13     }
14 }
```

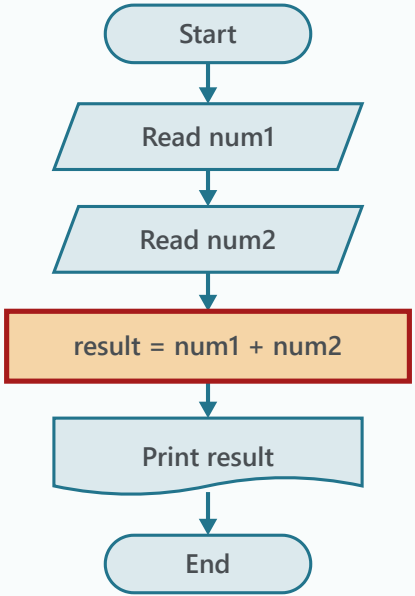
VARIABLES

`num1 = 7` `num2 = 5`

`result = 12`

OUTPUT

```
First number: 7
Second number: 5
```



Process box: `result = 7 + 5 = 12`.

From Flowchart to Java: The Sum Program

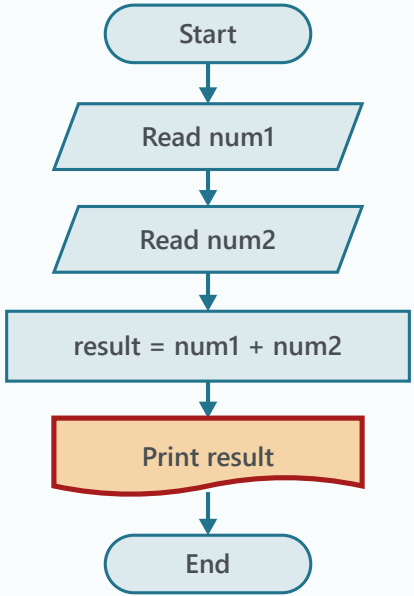
```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          System.out.print("First number: ");
7          int num1 = scanner.nextInt();
8          System.out.print("Second number: ");
9          int num2 = scanner.nextInt();
10         int result = num1 + num2;
▶          System.out.println("Result: " + result);
12         scanner.close();
13     }
14 }
```

VARIABLES

`num1 = 7` `num2 = 5` `result = 12`

OUTPUT

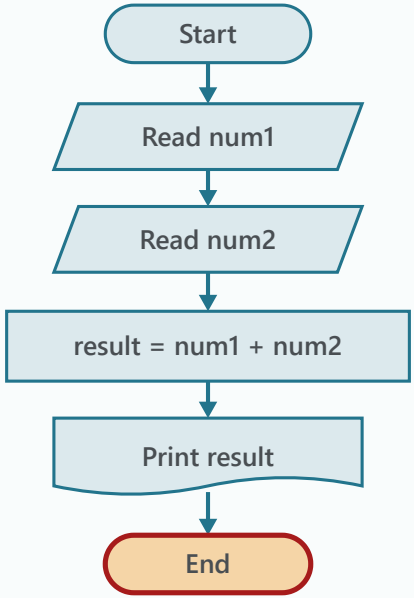
```
First number: 7
Second number: 5
Result: 12
```



Print box: `println` writes the result.

From Flowchart to Java: The Sum Program

```
1  import java.util.Scanner;
2
3  public class AddTwoNumbers {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          System.out.print("First number: ");
7          int num1 = scanner.nextInt();
8          System.out.print("Second number: ");
9          int num2 = scanner.nextInt();
10         int result = num1 + num2;
11         System.out.println("Result: " + result);
           scanner.close();
       }
14 }
```



VARIABLES

`num1 = 7` `num2 = 5` `result = 12`

OUTPUT

```
First number: 7
Second number: 5
Result: 12
```

The `Scanner` is closed and `main` ends (**End**). Every line ran exactly once.

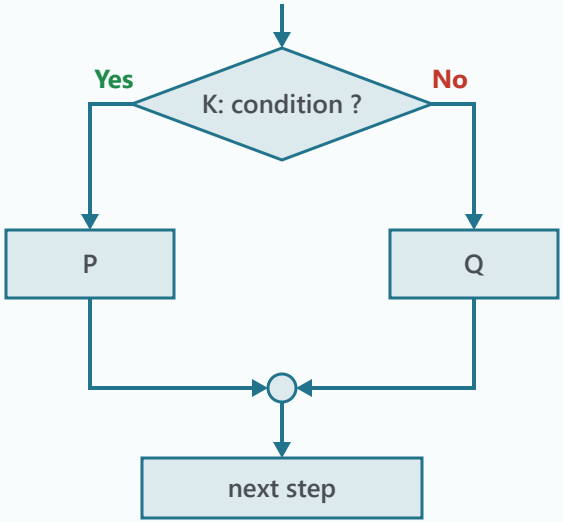
SELECTION

Conditional Flow

A flow in which **one** of two (or more) alternatives is chosen.

- **Decision point (K):** a condition whose answer is Yes or No.
- **Yes branch (P):** if the answer is yes, go on to P.
- **No branch (Q):** if the answer is no, go on to Q.
- It gives **flexibility** (different behavior in different situations), **decision making** (actions that depend on a condition) and **control** (check the state, act on it).

```
if (K) {
    P;   // condition true
} else {
    Q;   // condition false
}
```

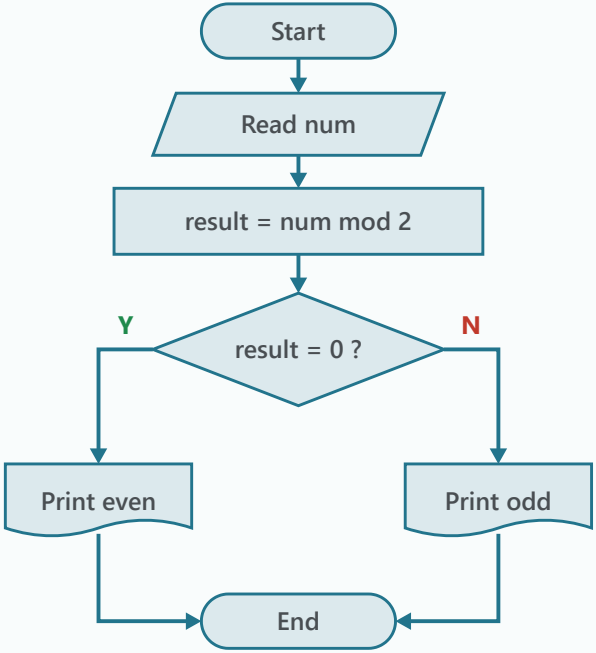


Example: Is the Number Odd or Even?

Problem: print whether a given number is odd or even.

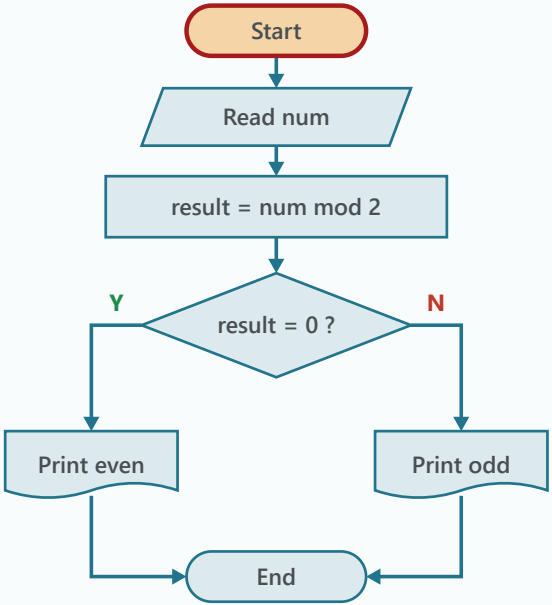
1. Start
2. Read the number
3. Compute number mod 2, assign to result
4. If result is 0 go to 5, else go to 6
5. Print "even", go to 7
6. Print "odd"
7. End

mod: the remainder. 7 mod 2 = 1, 8 mod 2 = 0. An even number leaves remainder 0 when divided by 2. In Java: `num % 2`.



From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
▶      public static void main(String[] args) {
▶          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
8          int result = num % 2;
9          if (result == 0) {
10             System.out.println("even");
11         } else {
12             System.out.println("odd");
13         }
14         scanner.close();
15     }
16 }
```



Start: a `Scanner` is created.

Step 1 / 6

VARIABLES

`num = –`

`result = –`

OUTPUT

From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
4      public static void main(String[] args) {
5          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
8          int result = num % 2;
9          if (result == 0) {
10             System.out.println("even");
11         } else {
12             System.out.println("odd");
13         }
14         scanner.close();
15     }
16 }
```

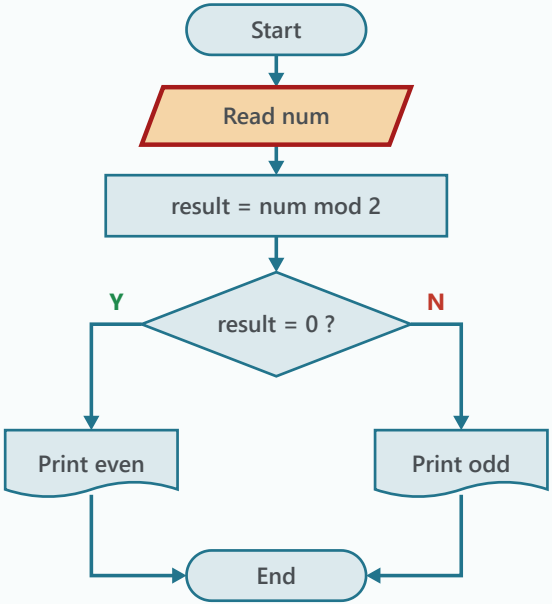
VARIABLES

`num = 7`

`result = –`

OUTPUT

```
Number: 7
```



The user enters **7**: `num = 7`.

From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
4      public static void main(String[] args) {
5          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
▶          int result = num % 2;
9          if (result == 0) {
10             System.out.println("even");
11         } else {
12             System.out.println("odd");
13         }
14         scanner.close();
15     }
16 }
```

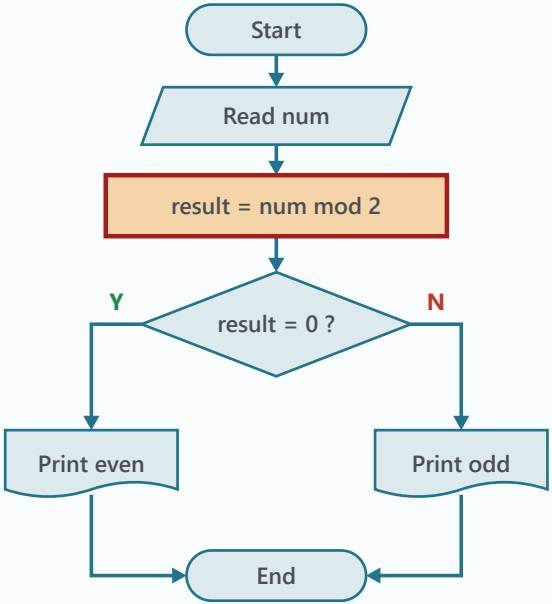
VARIABLES

`num = 7`

`result = 1`

OUTPUT

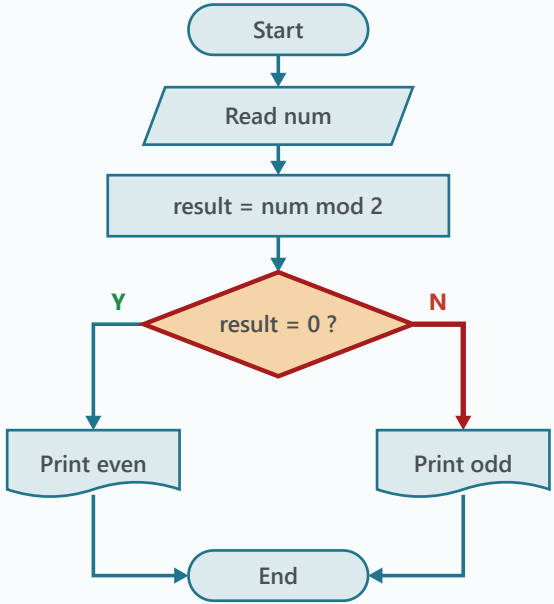
```
Number: 7
```



`result = 7 % 2 = 1` (7 = 3 × 2 + 1).

From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
4      public static void main(String[] args) {
5          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
8          int result = num % 2;
▶          if (result == 0) {
10             System.out.println("even");
11         } else {
12             System.out.println("odd");
13         }
14         scanner.close();
15     }
16 }
```



Decision: `1 == 0` is **false** → the No (N) path, the `else` block.

VARIABLES

`num = 7`

`result = 1`

OUTPUT

```
Number: 7
```

From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
4      public static void main(String[] args) {
5          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
8          int result = num % 2;
9          if (result == 0) {
10             System.out.println("even");
▶          } else {
▶              System.out.println("odd");
13         }
14         scanner.close();
15     }
16 }
```

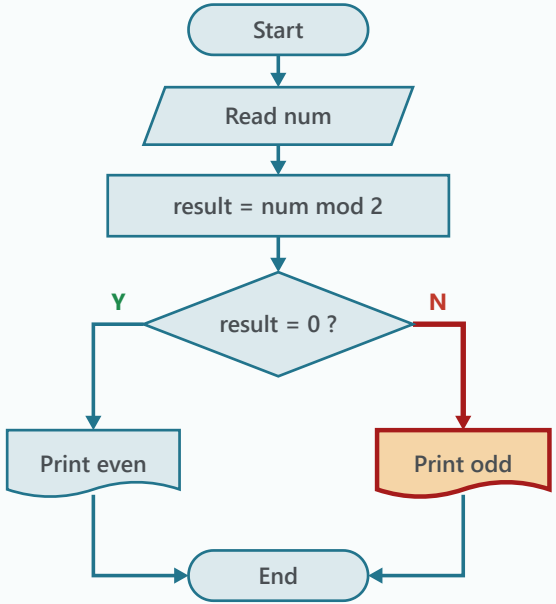
VARIABLES

`num = 7`

`result = 1`

OUTPUT

```
Number: 7
odd
```



"odd" is printed. Line 10 in the `if` block was **skipped**.

From Flowchart to Java: Odd or Even?

```
1  import java.util.Scanner;
2
3  public class OddOrEven {
4      public static void main(String[] args) {
5          Scanner scanner = new
   Scanner(System.in);
6          System.out.print("Number: ");
7          int num = scanner.nextInt();
8          int result = num % 2;
9          if (result == 0) {
10             System.out.println("even");
11         } else {
12             System.out.println("odd");
13         }
▶          scanner.close();
▶      }
16 }
```

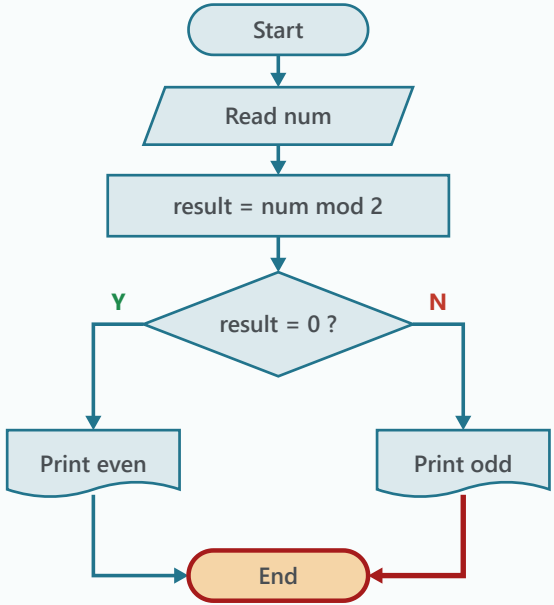
VARIABLES

`num = 7`

`result = 1`

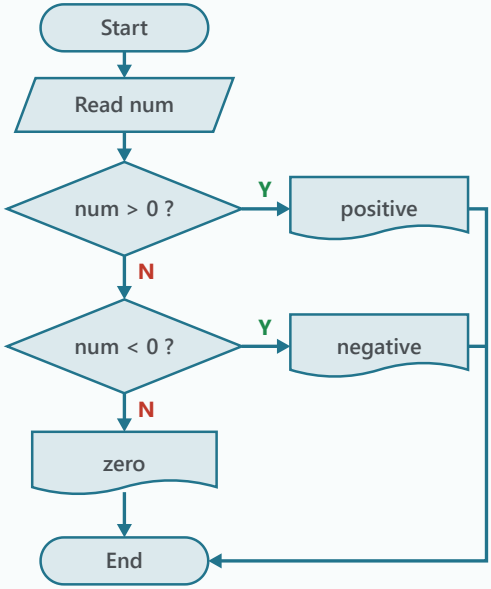
OUTPUT

```
Number: 7
odd
```



The two paths meet at **End**. With 8, `result = 0` and the Y path would print "even".

Nested Decisions: Positive, Negative, Zero



With three outcomes, chain two decision boxes: the first one's **No** exit leads to the second.

```
int num = scanner.nextInt();
if (num > 0) {
    System.out.println("positive");
} else if (num < 0) {
    System.out.println("negative");
} else {
    System.out.println("zero");
}
```

input	path taken	output
5	decision 1 Y	`positive`
−2	decision 1 N, decision 2 Y	`negative`
0	decision 1 N, decision 2 N	`zero`

One-way decision (Yes branch only): in Java, an `if` without `else`.

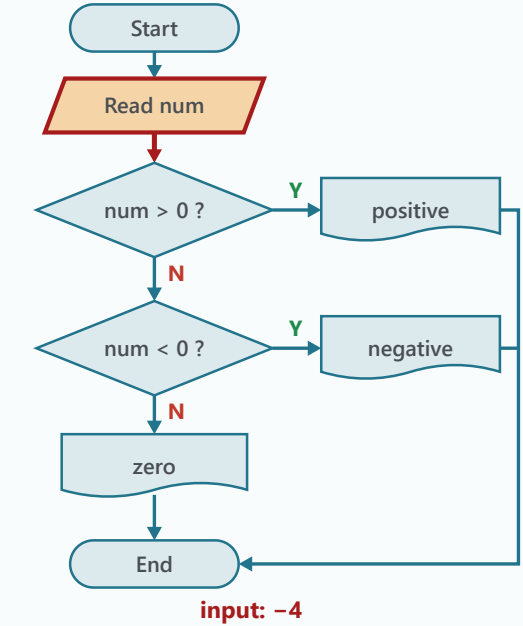
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
▶ int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = -4`

OUTPUT



Run 1: the user enters −4, `num` = −4.

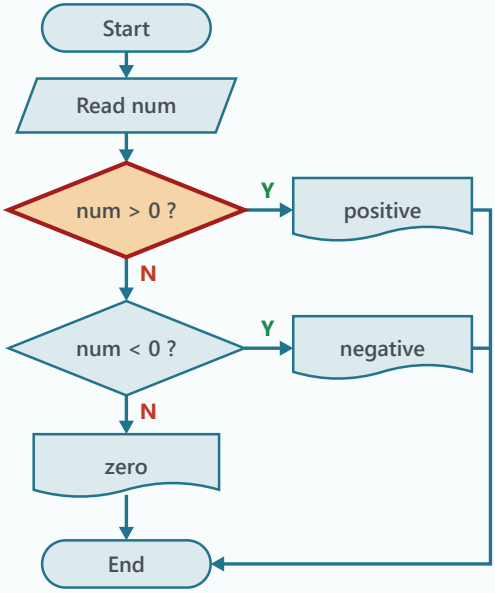
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
▶ if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = -4`

OUTPUT



−4 > 0 → false

Decision 1: `−4 > 0` is **false** → No exit.

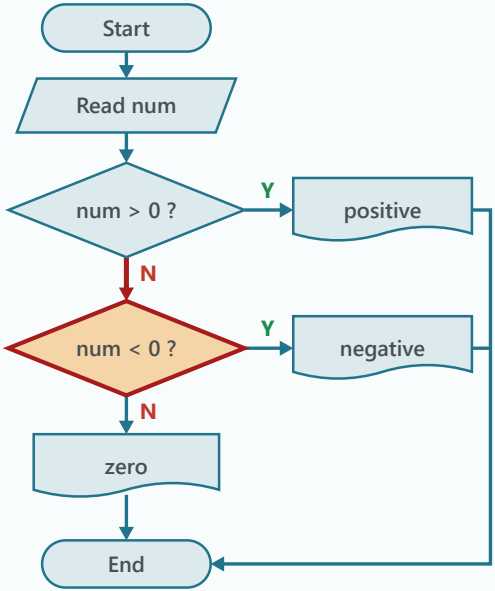
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
▶ } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = -4`

OUTPUT



−4 < 0 → true

The No exit leads to decision 2: `-4 < 0` is **true** → Yes.

Nested Decisions: Code Trace

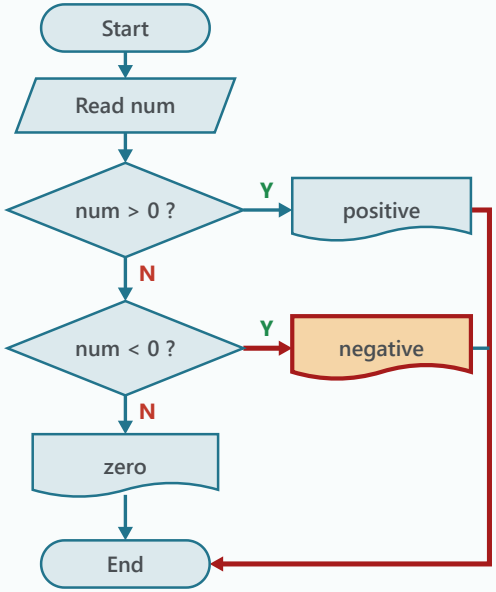
```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
▶     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = -4`

OUTPUT

```
negative
```



output: negative

`negative` is printed; the `else` branch is skipped and the flow goes to End.

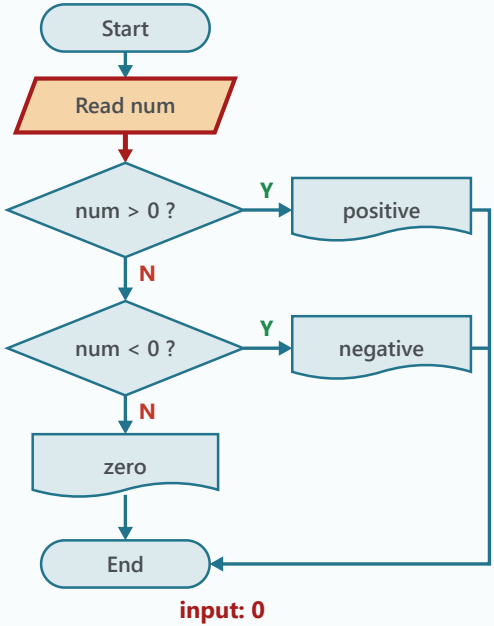
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
▶ int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = 0`

OUTPUT



Run 2: this time the user enters 0 (the output panel shows the new run).

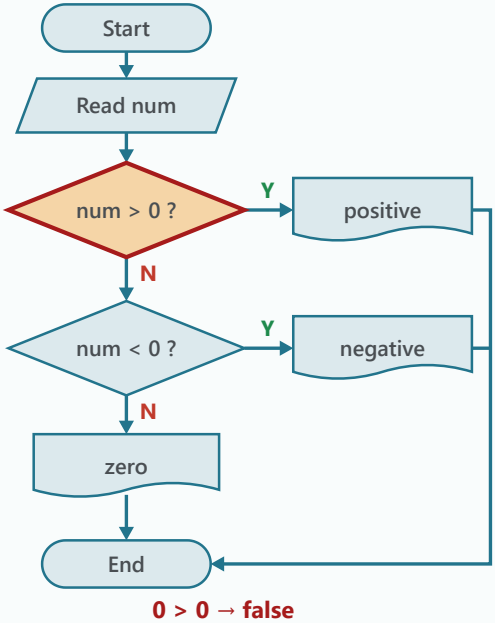
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
▶ if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = 0`

OUTPUT



0 > 0 → false

`0 > 0` is **false**: 0 is not positive.

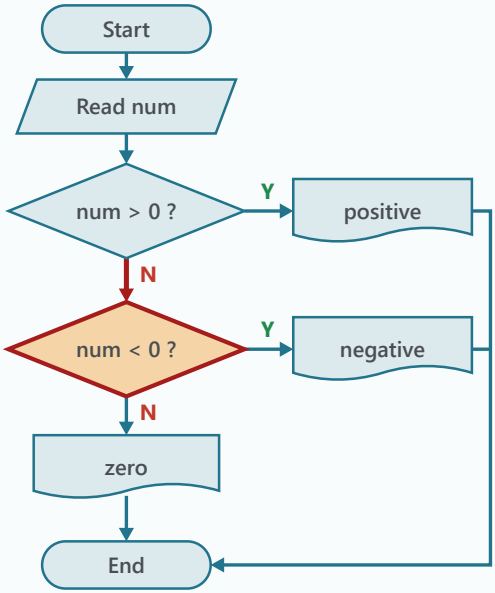
Nested Decisions: Code Trace

```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
▶ } else if (num < 0) {
6     System.out.println("negative");
7 } else {
8     System.out.println("zero");
9 }
```

VARIABLES

`num = 0`

OUTPUT



0 < 0 → false

`0 < 0` is **false** too → the second decision's No exit.

Nested Decisions: Code Trace

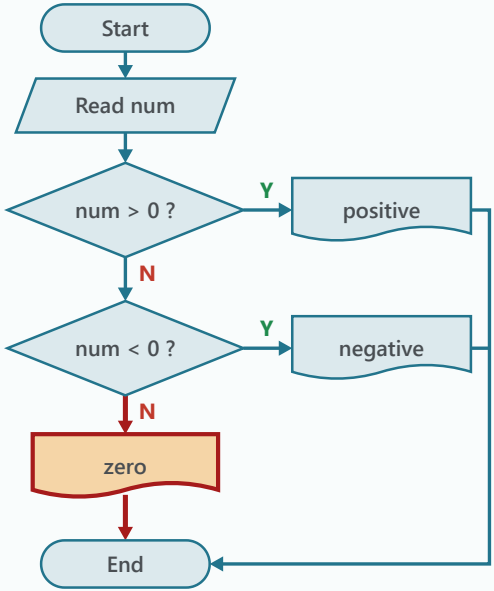
```
1 Scanner scanner = new Scanner(System.in);
2 int num = scanner.nextInt();
3 if (num > 0) {
4     System.out.println("positive");
5 } else if (num < 0) {
6     System.out.println("negative");
7 } else {
▶     System.out.println("zero");
9 }
```

VARIABLES

`num = 0`

OUTPUT

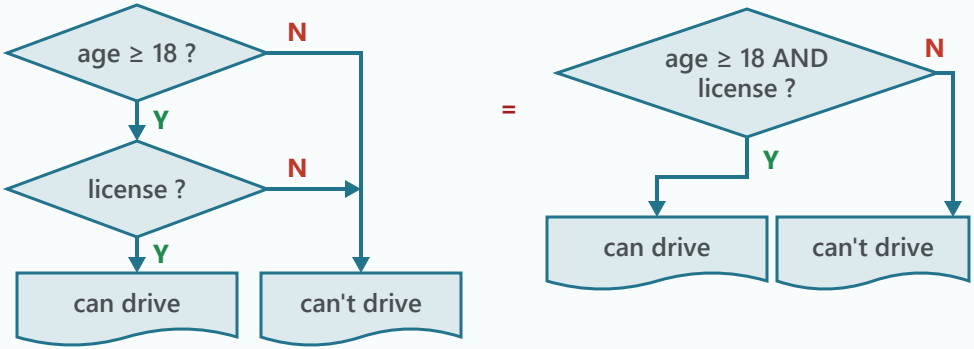
```
zero
```



output: zero

Both conditions are false: the last `else` runs and prints `zero`. Exactly **one** of the three branches runs each time.

Compound Conditions: AND, OR



age	license	age ≥ 18 AND license	age ≥ 18 OR license
20	yes	Yes → can drive	Yes
20	no	No → can't drive	Yes
16	yes	No → can't drive	Yes
16	no	No → can't drive	No

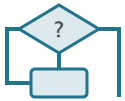
- **AND**: Yes only if both conditions are true. Drawn as two decisions, the first one's **No** exit goes straight to "can't drive"; the second is never asked.
- **OR**: one true condition is enough. Drawn as two decisions, the first one's **Yes** exit goes straight to the result.
- Java: `if (age >= 18 && license)` , `||` . Java, too, checks the second condition only when needed.

ITERATION

Iteration (Loops)

A flow in which steps repeat **more than once**. In the chart, an arrow leads back to the decision box.

Condition-controlled loop



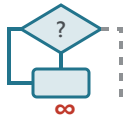
Repeats while a condition holds. The number of rounds may be unknown in advance (e.g. read until the user enters 0). Java: `while`.

Counted loop



The number of repetitions is known up front (e.g. 1 to 10). Java: `for`.

Infinite loop



The condition always holds, so the steps repeat forever. Usually a **bug**; used on purpose in always-on programs such as servers.

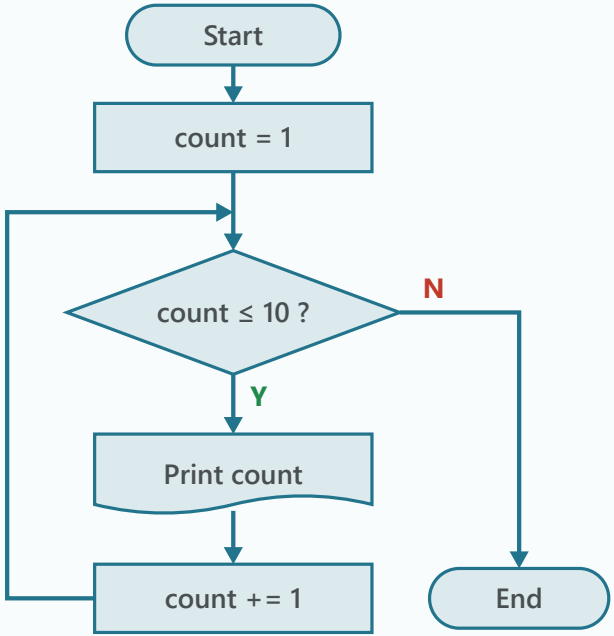
- + **Efficient:** the same operation runs as often as needed without rewriting the code.
- + **Dynamic:** the number of repetitions can depend on the input; the program adapts.
- + **Avoids repeated code:** printing 1000 numbers takes 3–4 lines, not 1000.

Example: Printing the Numbers 1 to 10

Algorithm

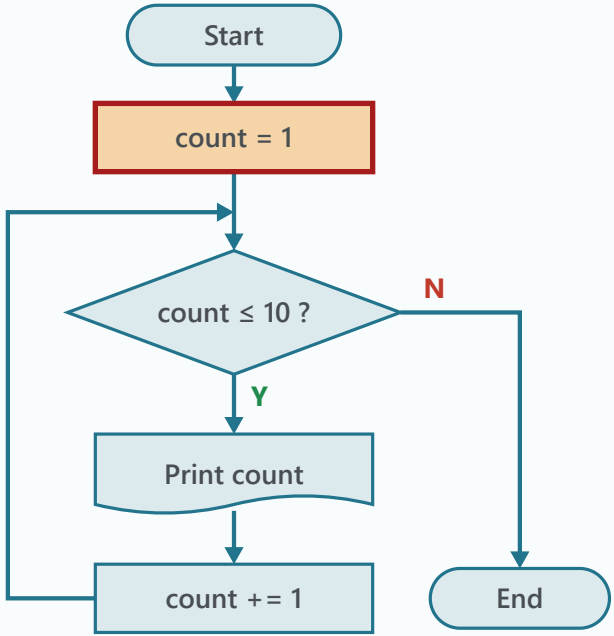
1. Start
2. count = 1
3. If count ≤ 10 go to 4, else go to 7
4. Print count
5. Add 1 to count
6. Go to 3
7. End

Step 6, "Go to 3", is the **back arrow** returning to the top of the decision box. That arrow makes the loop; the decision box is its exit door.



From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
          int count = 1;
4         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

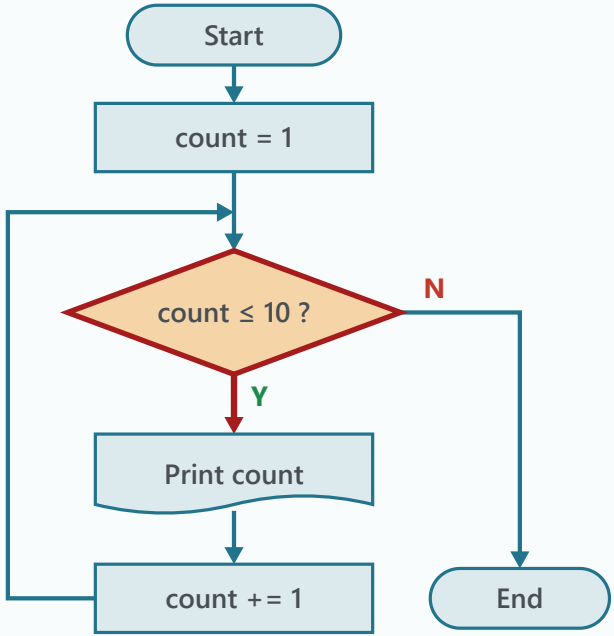
`count = 1`

OUTPUT

`count = 1` : the loop variable gets its initial value.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 1`

OUTPUT

Condition: `1 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

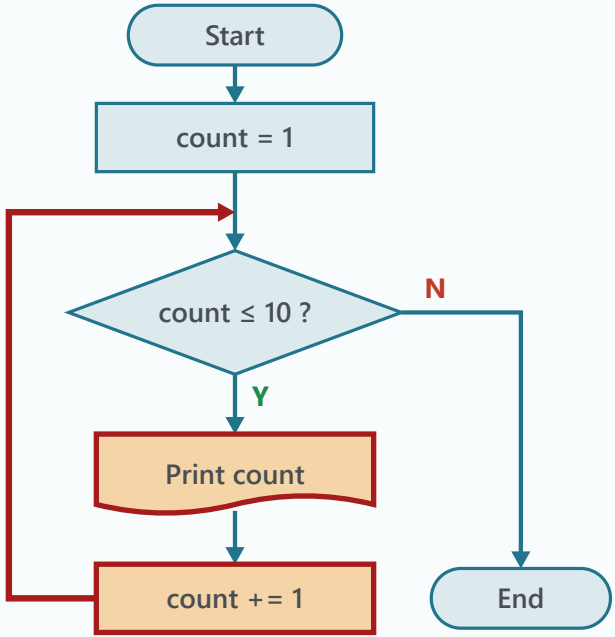
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 2`

OUTPUT

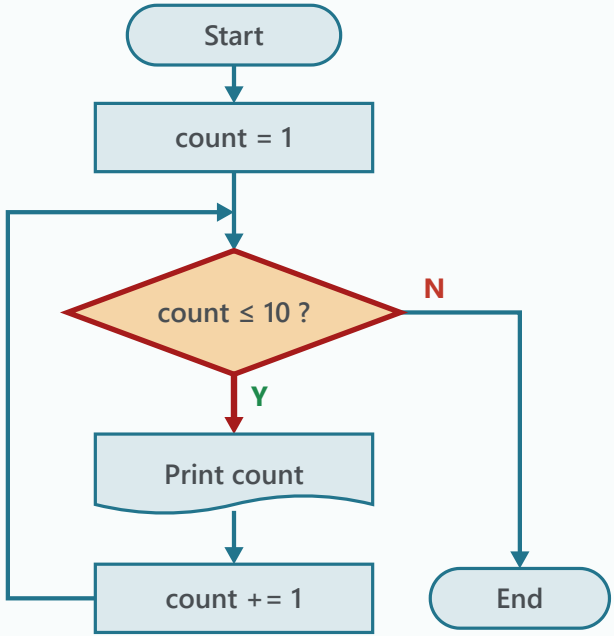
```
1
```



1 is printed and `count` becomes 2; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 2`

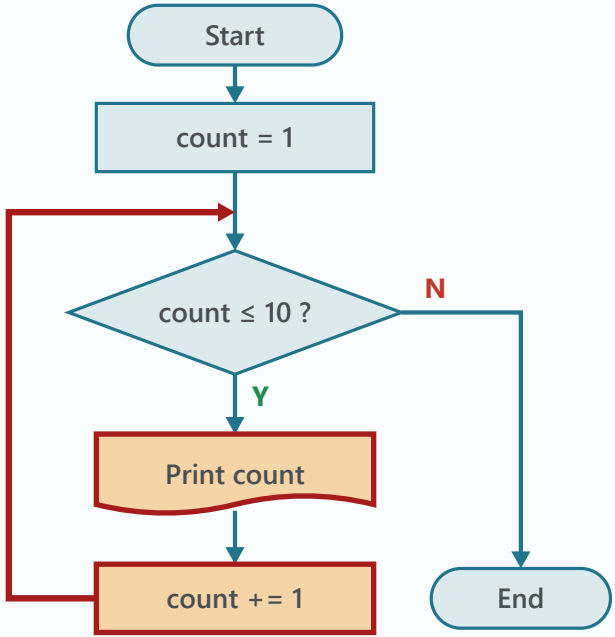
OUTPUT

```
1
```

Condition: `2 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 3`

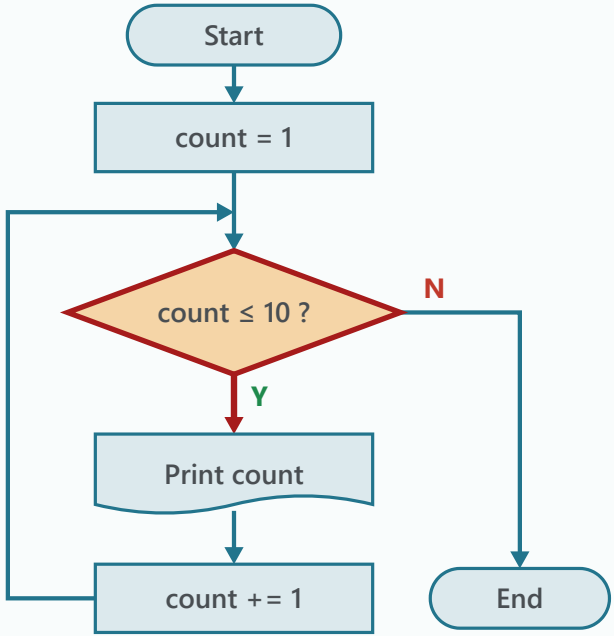
OUTPUT

```
1 2
```

2 is printed and `count` becomes 3; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 3`

OUTPUT

```
1 2
```

Condition: `3 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

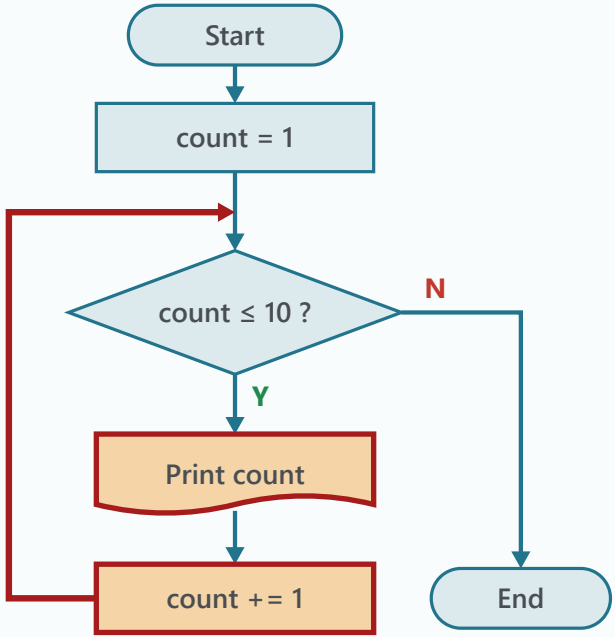
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 4`

OUTPUT

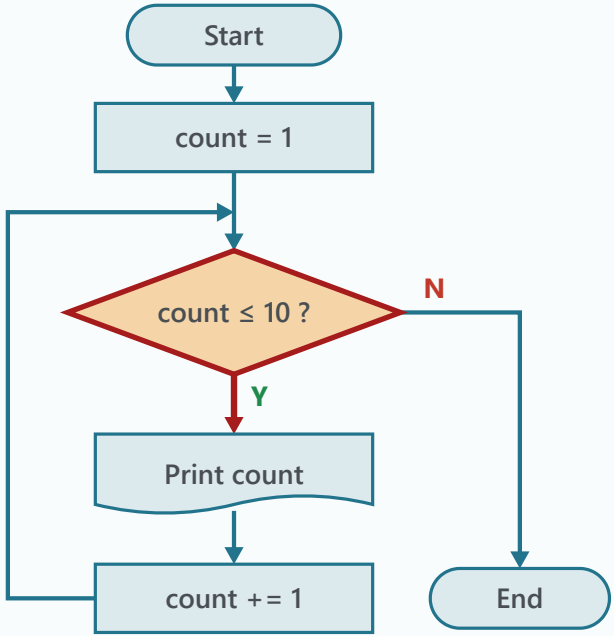
```
1 2 3
```



3 is printed and `count` becomes 4; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 4`

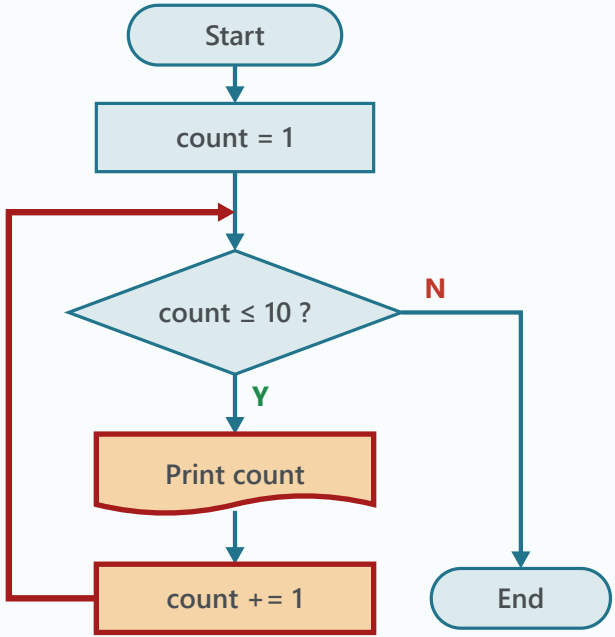
OUTPUT

```
1 2 3
```

Condition: `4 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 5`

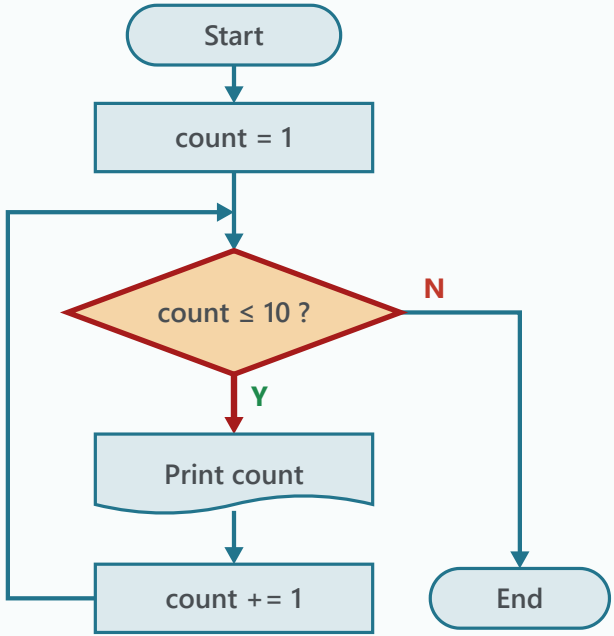
OUTPUT

```
1 2 3 4
```

4 is printed and `count` becomes 5; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 5`

OUTPUT

```
1 2 3 4
```

Condition: `5 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

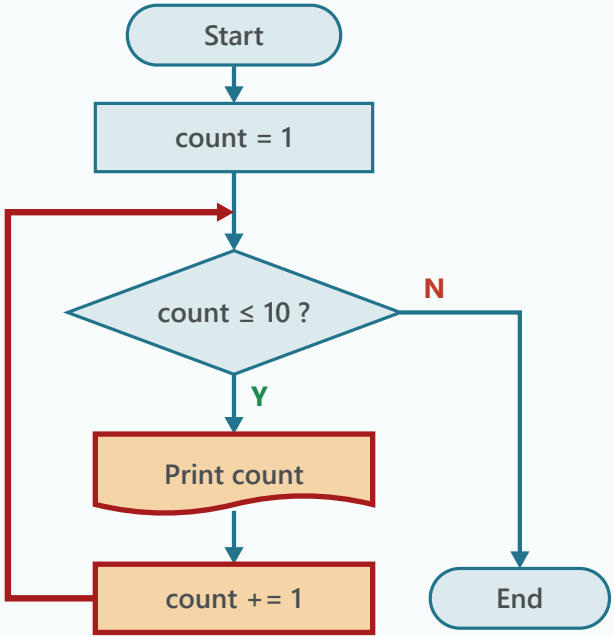
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 6`

OUTPUT

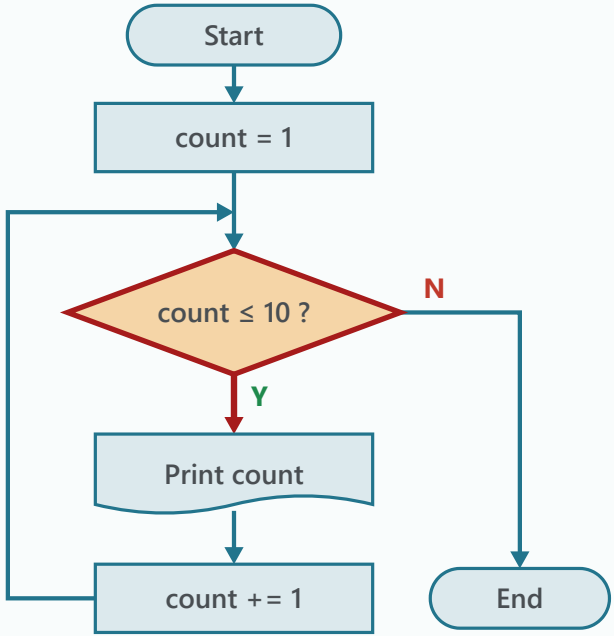
```
1 2 3 4 5
```



5 is printed and `count` becomes 6; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 6`

OUTPUT

```
1 2 3 4 5
```

Condition: `6 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

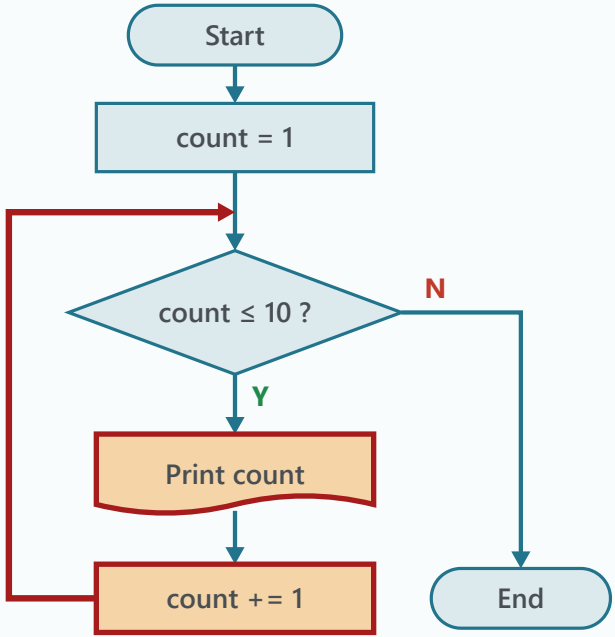
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 7`

OUTPUT

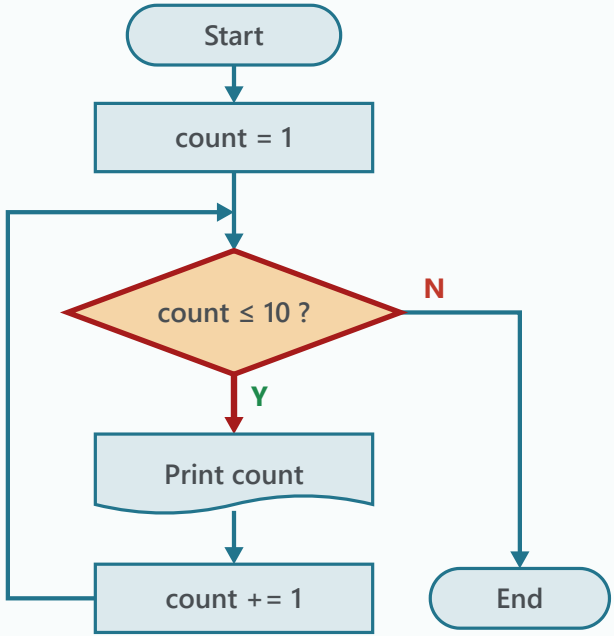
```
1 2 3 4 5 6
```



6 is printed and `count` becomes 7; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 7`

OUTPUT

```
1 2 3 4 5 6
```

Condition: `7 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

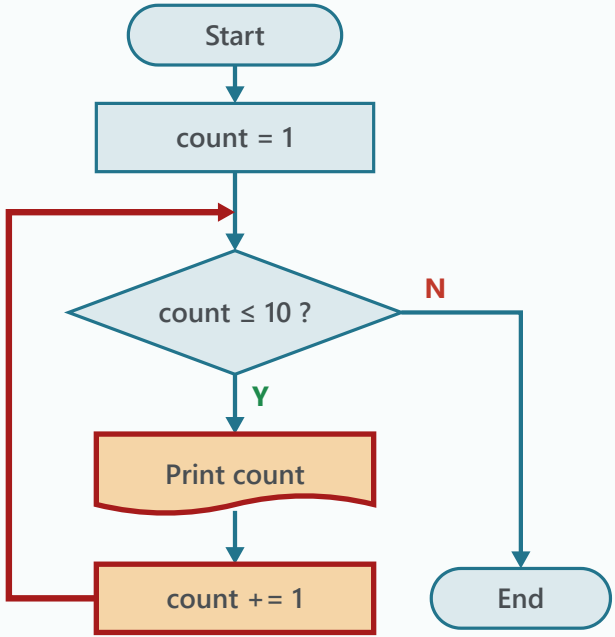
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 8`

OUTPUT

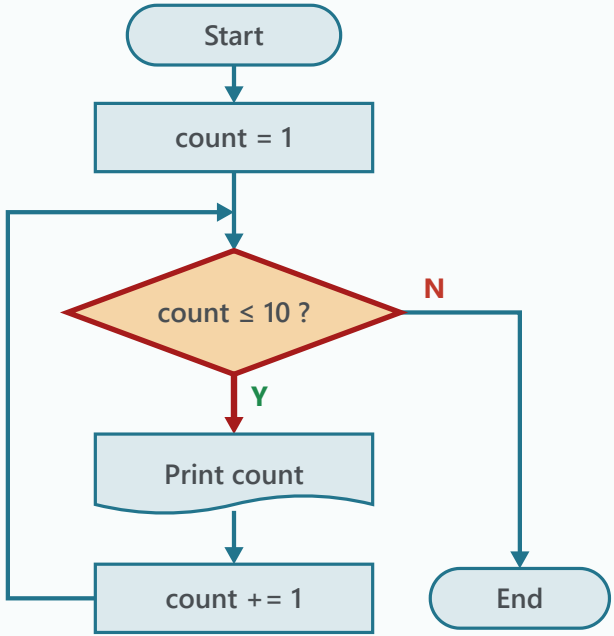
```
1 2 3 4 5 6 7
```



7 is printed and `count` becomes 8; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 8`

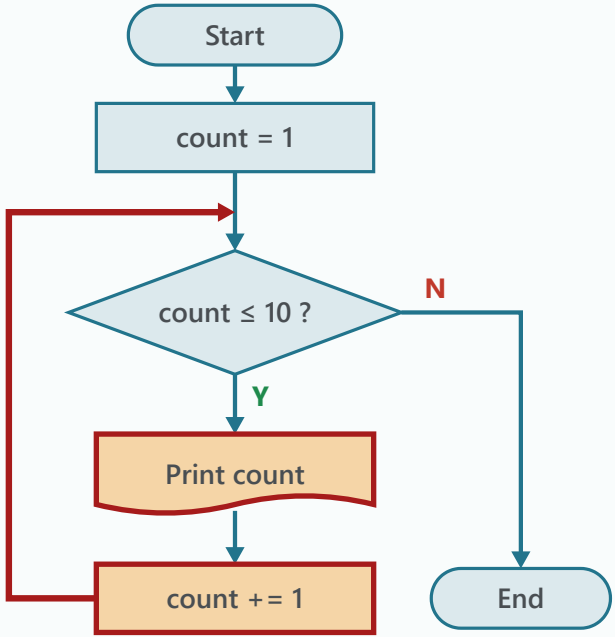
OUTPUT

```
1 2 3 4 5 6 7
```

Condition: `8 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 9`

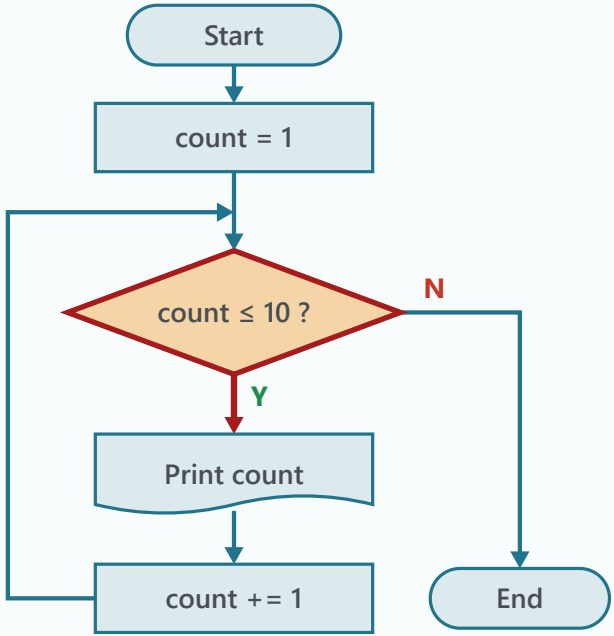
OUTPUT

```
1 2 3 4 5 6 7 8
```

8 is printed and `count` becomes 9; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 9`

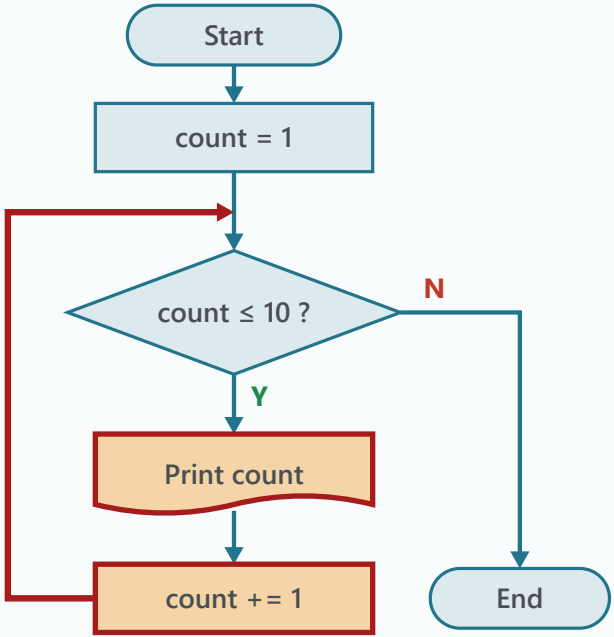
OUTPUT

```
1 2 3 4 5 6 7 8
```

Condition: `9 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 10`

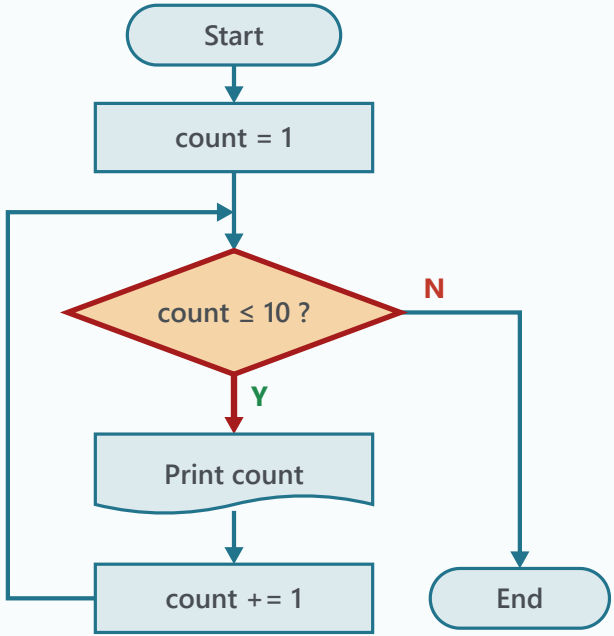
OUTPUT

```
1 2 3 4 5 6 7 8 9
```

9 is printed and `count` becomes 10; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 10`

OUTPUT

```
1 2 3 4 5 6 7 8 9
```

Condition: `10 <= 10` is **true** → Yes, enter the loop body.

From Flowchart to Java: the while Loop

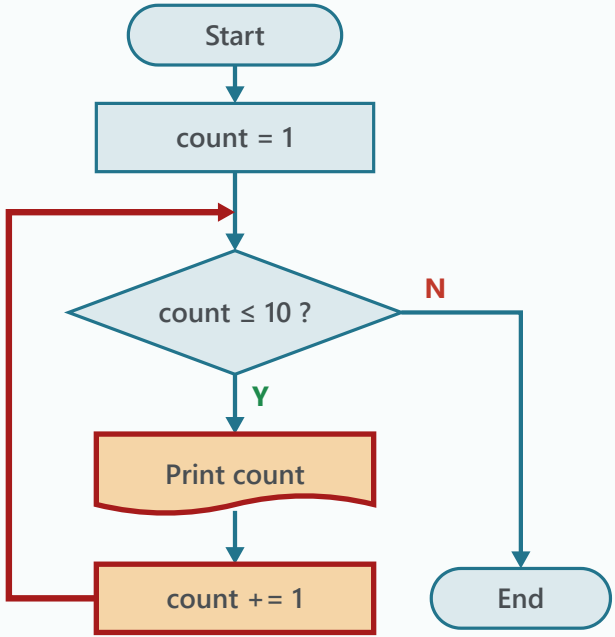
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
              System.out.print(count + " ");
              count += 1;
7         }
8     }
9 }
```

VARIABLES

`count = 11`

OUTPUT

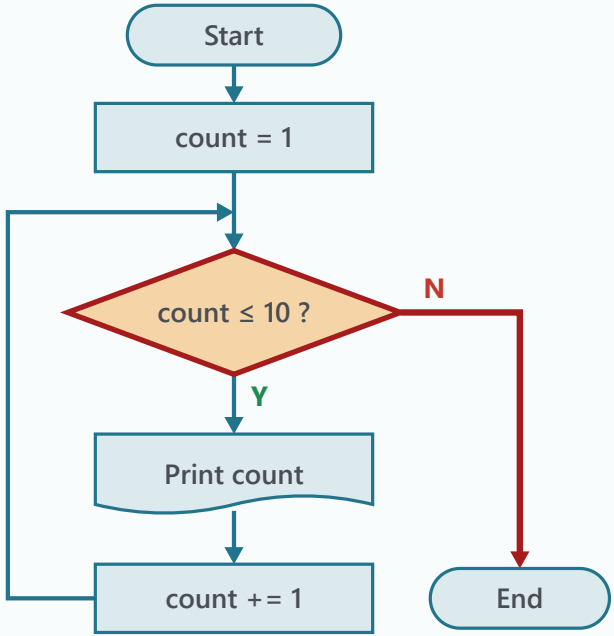
```
1 2 3 4 5 6 7 8 9
10
```



10 is printed and `count` becomes 11; the back arrow returns to the condition.

From Flowchart to Java: the while Loop

```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
▶         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
8     }
9 }
```



VARIABLES

`count = 11`

OUTPUT

```
1 2 3 4 5 6 7 8 9
10
```

Condition: `11 <= 10` is **false** → No, leave the loop.

From Flowchart to Java: the while Loop

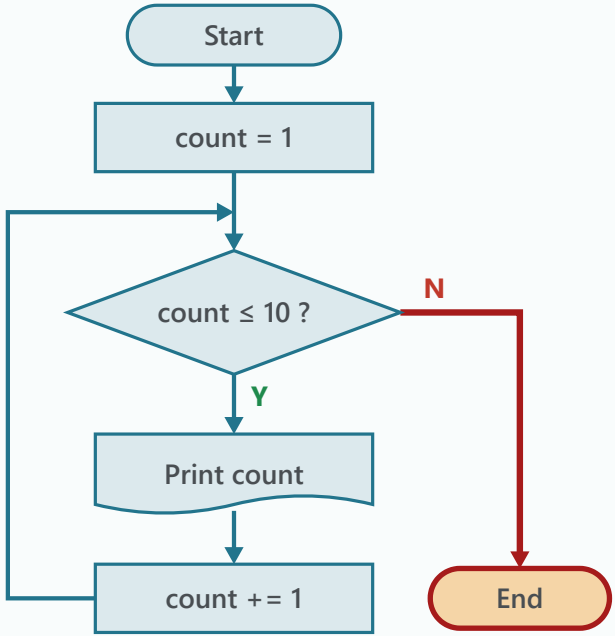
```
1 public class OneToTen {
2     public static void main(String[] args)
  {
3         int count = 1;
4         while (count <= 10) {
5             System.out.print(count + " ");
6             count += 1;
7         }
      }
9 }
```

VARIABLES

`count = 11`

OUTPUT

```
1 2 3 4 5 6 7 8 9
10
```



End. The body ran 10 times, the condition 11 times.

Boundary Condition: < or ≤?

A single character in the loop condition changes the number of rounds. The counter starts at 1 and goes up by 1 each round:

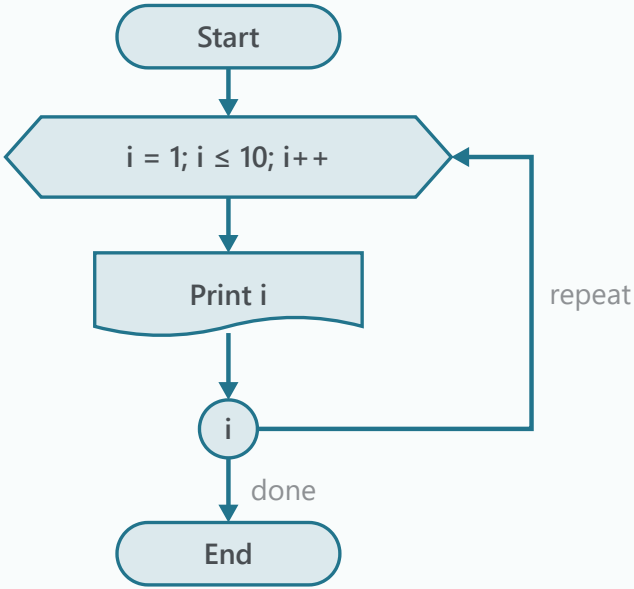
Condition	Body runs	Output	Prints 1 to 10?
`count < 10`	9	`1 2 3 4 5 6 7 8 9`	No: 10 missing
`count <= 10`	10	`1 2 3 4 5 6 7 8 9 10`	**Yes**
`count < 11`	10	`1 2 3 4 5 6 7 8 9 10`	**Yes** (same meaning)
`count <= 11`	11	`1 2 … 10 11`	No: extra 11

This is an **off-by-one error**. To avoid it, test the condition with the **last value**: is it true when the counter is 10? "1 **through** 10" includes 10, so the condition must be `count ≤ 10`.

A Dedicated Loop Flow (the Loop Symbol)

The same problem: print the numbers 1 to 10.

1. Start
2. Loop: 1 to 10
3. Print the loop variable
4. End of loop
5. End

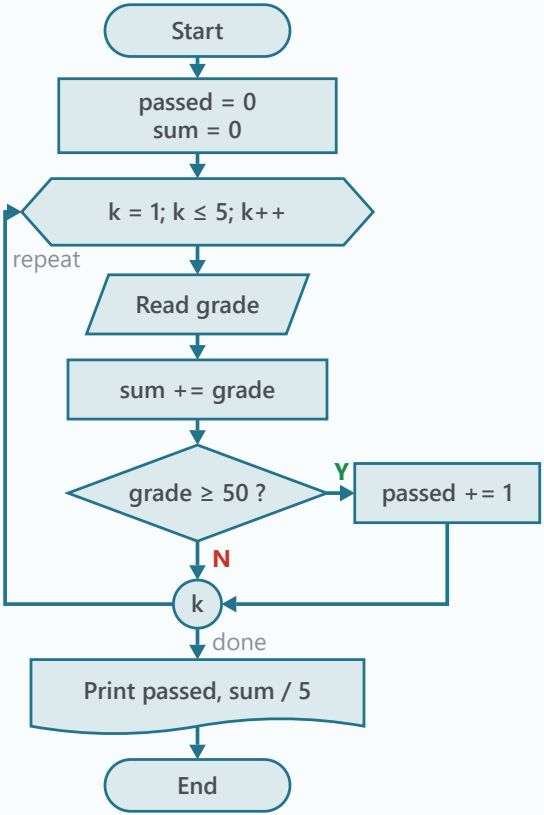


The hexagon gathers the three parts of the loop in one box:

- **initialization:** `i = 1` (once)
- **condition:** `i ≤ 10` (before each round)
- **update:** `i++`, i.e. add 1 to i (after each round)

The small circle marked "i" is the **end of the loop**: the flow returns from there to the hexagon, and goes on to End when the loop is done.

A Decision Inside a Loop: Counter and Accumulator

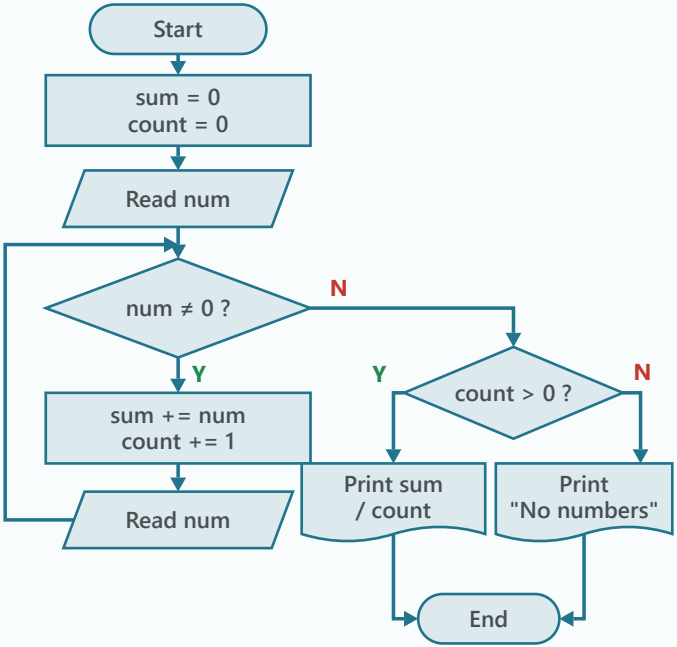



```
int passed = 0, sum = 0;
for (int k = 1; k <= 5; k++) {
  int grade = scanner.nextInt();
  sum += grade;
  if (grade >= 50) {
    passed++;
  }
}
System.out.println(passed + " passed, average "
    + sum / 5.0);
```

Input 45 70 50 30 88 → **Output** `3 passed, average 56.6`

- **Counter** (`passed`): goes up by 1 when the condition holds. **Accumulator** (`sum`): gains a value every round.
- The decision is **inside** the loop: it is tested again with each new grade. The No branch skips the counter and goes to the end of the loop.
- 50 is on the boundary: with `>=` a 50 passes; with `>` the count would be 2.
- `sum / 5.0` : with `5` , integer division would give 56.
- A third pattern is the **flag**: a `boolean` variable that records whether something happened (`found = true`).

Sentinel-Controlled Loops



- **Sentinel value**: a special value that marks the end of input and is not counted as data (0, −1, "end").
- The number of values is unknown in advance: the loop is condition-controlled, a `while` in Java.
- **Priming read:** read the first value before the loop and the next one at the **end** of the body. Then the condition sees the sentinel at once and it never joins the sum.
- Pick a sentinel that cannot occur in real data: for grades −1 works, 0 does not (a student may score 0).
- After the loop, check `count = 0` separately: if no numbers were entered, `sum / count` divides by zero (an `ArithmeticException` for integers, `NaN` for `double`).

Sentinel Loop: Code Trace

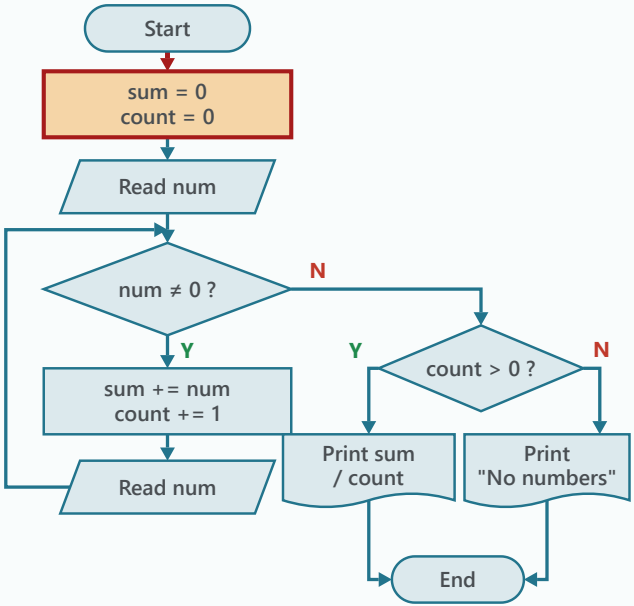
```
▶ int sum = 0, count = 0;
2 int num = scanner.nextInt();
3 while (num != 0) {
4   sum += num;
5   count++;
6   num = scanner.nextInt();
7 }
8 if (count > 0) {
9   System.out.println("Average: "
10      + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

`sum = 0`

`count = 0`

OUTPUT



sum = 0, count = 0

The accumulator and counter start at zero.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
▶ int num = scanner.nextInt();
3  while (num != 0) {
4    sum += num;
5    count++;
6    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

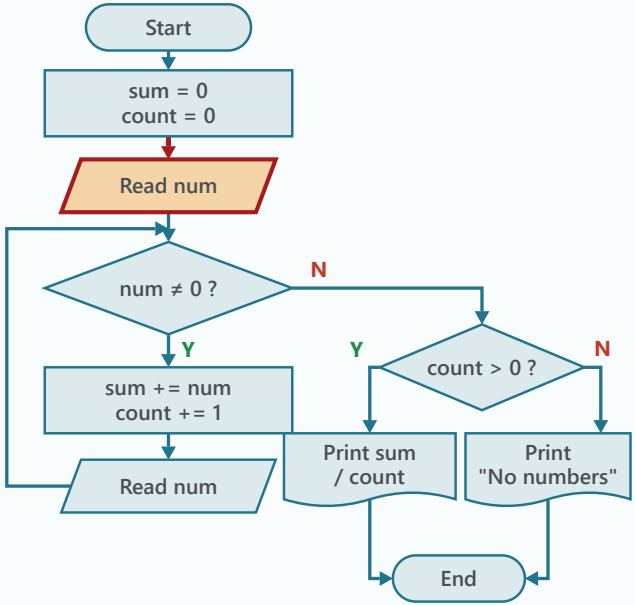
`sum = 0`

`count = 0`

`num = 12`

OUTPUT

```
12
```



num = 12

Priming read: the first number is read before the loop. Input: 12 7 5 0.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
▶  while (num != 0) {
4    sum += num;
5    count++;
6    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

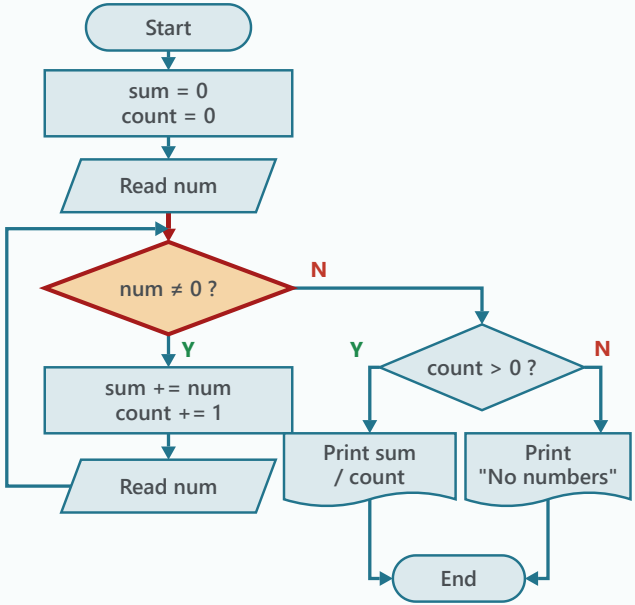
`sum = 0`

`count = 0`

`num = 12`

OUTPUT

```
12
```



12 ≠ 0 → Y

Condition: 12 ≠ 0 is **true** → Yes, enter the body.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
3  while (num != 0) {
▶    sum += num;
▶    count++;
▶    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

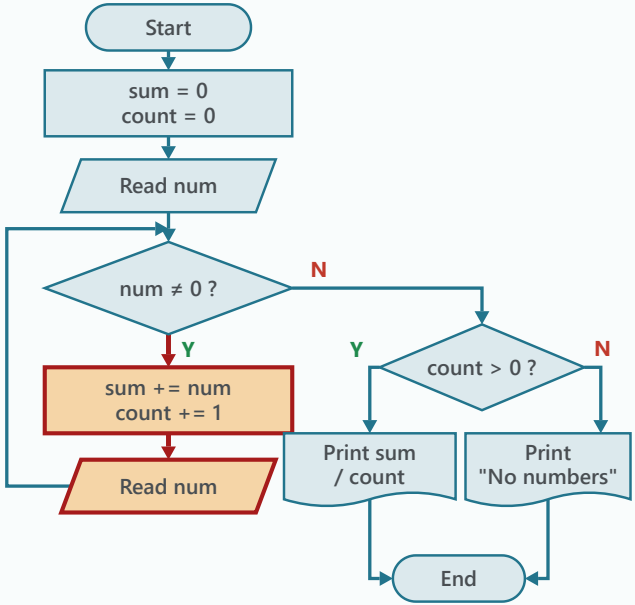
`sum = 12`

`count = 1`

`num = 7`

OUTPUT

```
12
7
```



sum = 12, count = 1, num = 7

`sum` = 12, `count` = 1; the next number is read at the end of the body: 7.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
▶  while (num != 0) {
▶    sum += num;
▶    count++;
▶    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

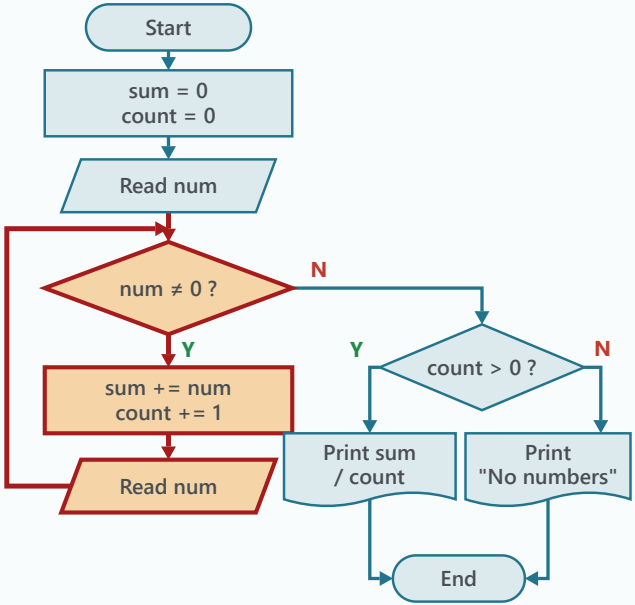
`sum = 19`

`count = 2`

`num = 5`

OUTPUT

```
12
7
5
```



7 ≠ 0 → sum = 19, count = 2

Back to the decision: 7 ≠ 0. `sum` = 19, `count` = 2; reads 5.

Sentinel Loop: Code Trace

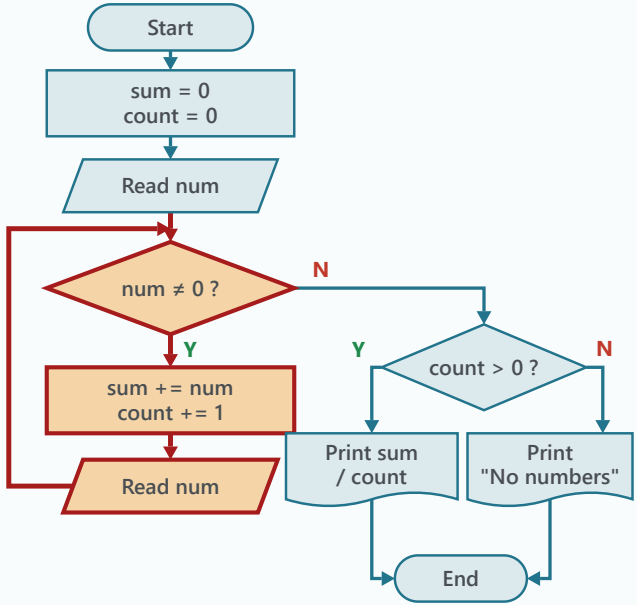
```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
▶  while (num != 0) {
▶    sum += num;
▶    count++;
▶    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

- `sum = 24`
- `count = 3`
- `num = 0`

OUTPUT

```
12
7
5
0
```



5 ≠ 0 → sum = 24, count = 3

5 ≠ 0: `sum` = 24, `count` = 3; reads 0, the sentinel.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
▶  while (num != 0) {
4    sum += num;
5    count++;
6    num = scanner.nextInt();
7  }
8  if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

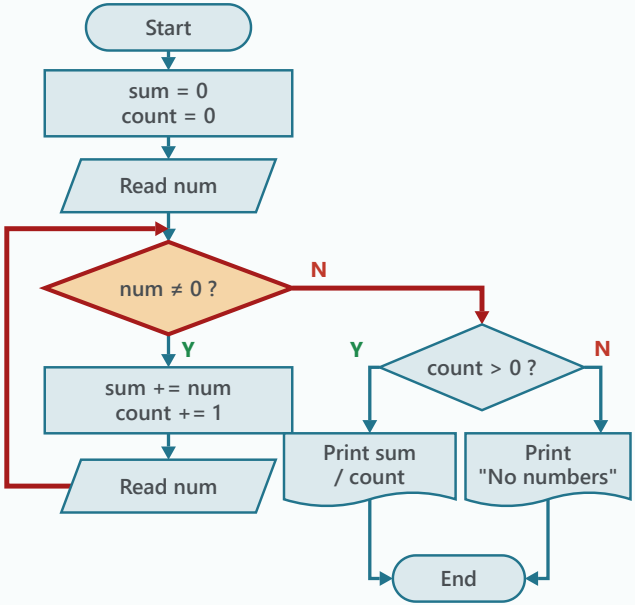
`sum = 24`

`count = 3`

`num = 0`

OUTPUT

```
12
7
5
0
```



0 ≠ 0 → N

0 ≠ 0 is **false** → No: leave the loop. The 0 never joined the sum.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
3  while (num != 0) {
4    sum += num;
5    count++;
6    num = scanner.nextInt();
7  }
▶ if (count > 0) {
9    System.out.println("Average: "
10       + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

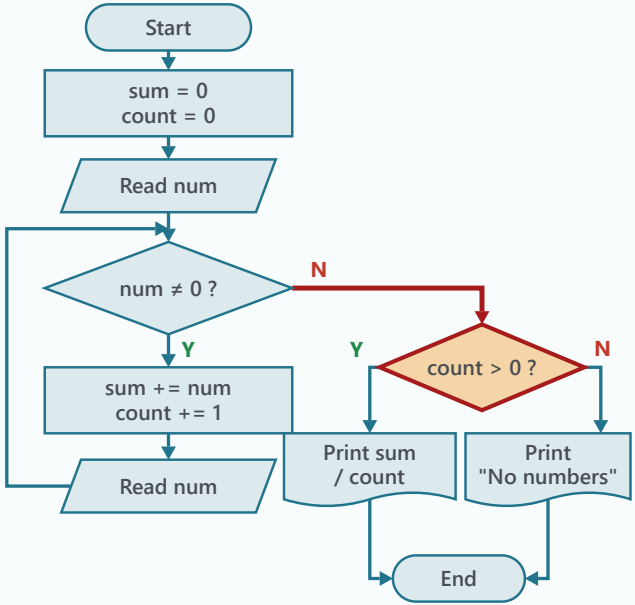
`sum = 24`

`count = 3`

`num = 0`

OUTPUT

```
12
7
5
0
```



3 > 0 → Y

Check after the loop: `count` = 3 > 0 → Yes.

Sentinel Loop: Code Trace

```
1  int sum = 0, count = 0;
2  int num = scanner.nextInt();
3  while (num != 0) {
4    sum += num;
5    count++;
6    num = scanner.nextInt();
7  }
8  if (count > 0) {
▶    System.out.println("Average: "
▶        + (double) sum / count);
11 } else {
12   System.out.println("No numbers");
13 }
```

VARIABLES

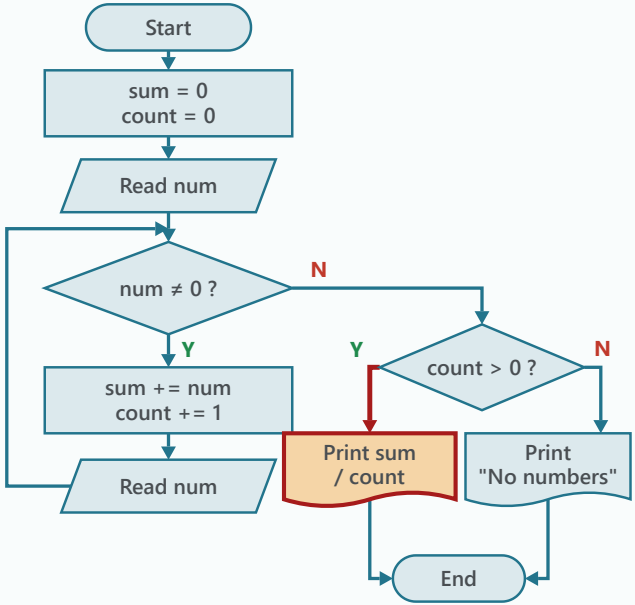
`sum = 24`

`count = 3`

`num = 0`

OUTPUT

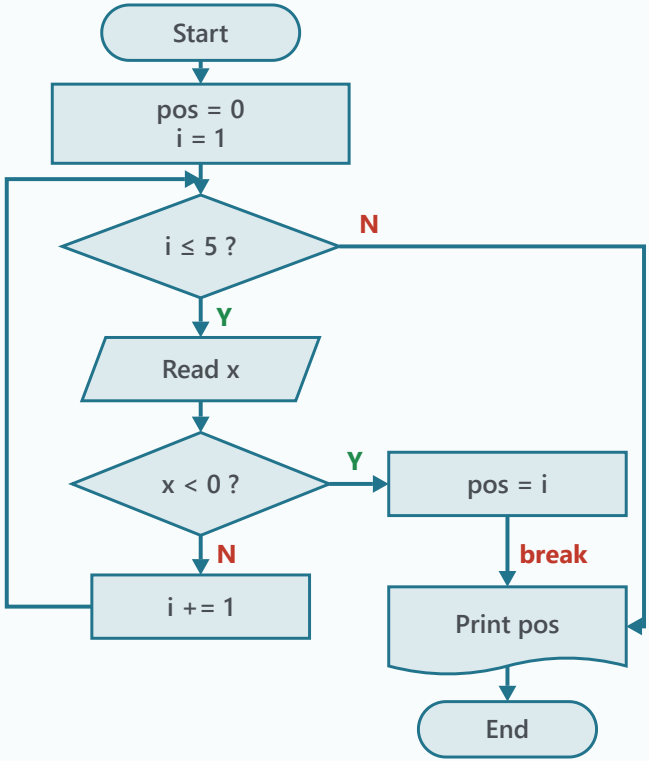
```
12
7
5
0
Average: 8.0
```



24 / 3 = 8.0

24 / 3 = 8.0 is printed. Had the first number been 0, the No branch would print `No numbers`.

Leaving a Loop Early



- Once a search finds its target, the remaining rounds are pointless: a **second decision** inside the loop takes the flow out of it.
- The loop has two exits in the chart: when the condition turns false (not found) and when the target is found. Both paths meet at the **Print pos** box.
- In Java the second exit is `break`; it ends only the loop it is in.
- `pos = 0` means "not found"; positions start at 1, so it cannot be mistaken for a real one.
- The structured, single-exit version uses a **flag**:

```
int pos = 0;
boolean found = false;
for (int i = 1; i <= 5 && !found; i++) {
  int x = scanner.nextInt();
  if (x < 0) {
    pos = i;
    found = true;
  }
}
```

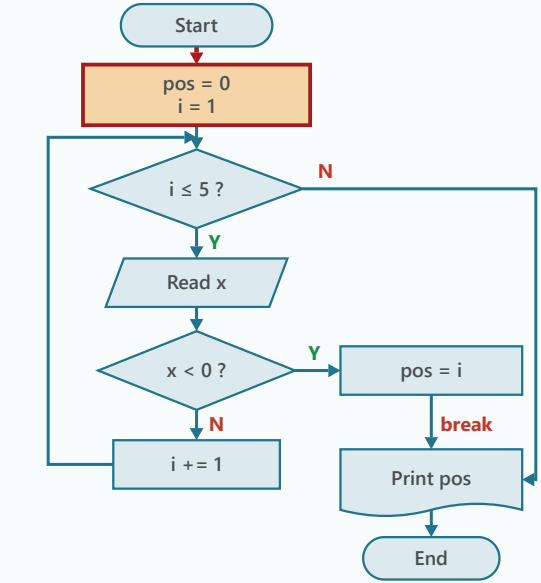
Early Exit: Code Trace

```
▶ int pos = 0;
▶ for (int i = 1; i <= 5; i++) {
3   int x = scanner.nextInt();
4   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

VARIABLES

`pos = 0` `i = 1`

OUTPUT



pos = 0, i = 1

`pos` = 0 (none found yet), `i` = 1. Input: 4 9 −2 6 1.

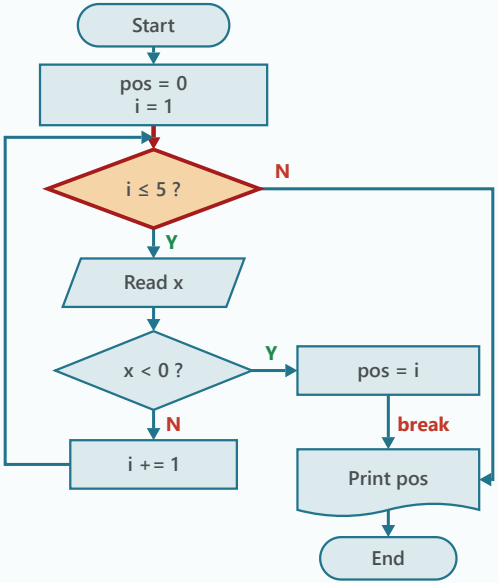
Early Exit: Code Trace

```
1 int pos = 0;
▶ for (int i = 1; i <= 5; i++) {
3   int x = scanner.nextInt();
4   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

VARIABLES

`pos = 0` `i = 1`

OUTPUT



1 ≤ 5 → Y

1 ≤ 5 → Yes, enter the body.

Early Exit: Code Trace

```
1 int pos = 0;
2 for (int i = 1; i <= 5; i++) {
▶   int x = scanner.nextInt();
▶   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

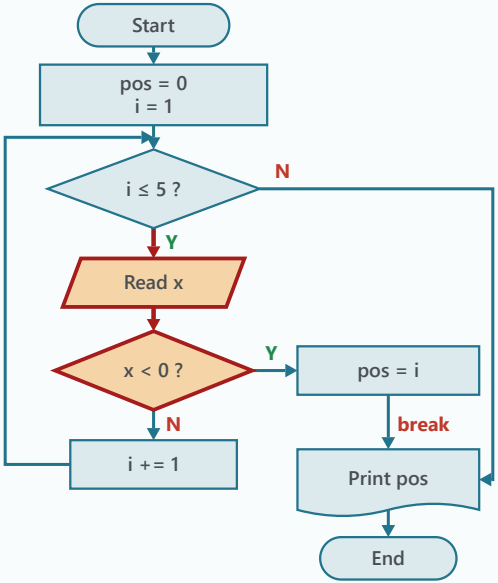
VARIABLES

`pos = 0` `i = 1`

`x = 4`

OUTPUT

```
4
```



x = 4: 4 < 0 → N

`x` = 4; 4 < 0 is false → No.

Early Exit: Code Trace

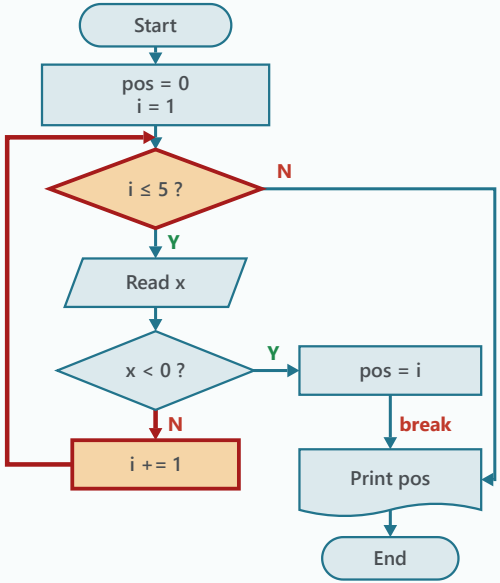
```
1 int pos = 0;
▶ for (int i = 1; i <= 5; i++) {
3   int x = scanner.nextInt();
4   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

VARIABLES

`pos = 0` `i = 2`

OUTPUT

```
4
```



i = 2: 2 ≤ 5 → Y

`i` = 2; back to the decision: 2 ≤ 5 → Yes.

Early Exit: Code Trace

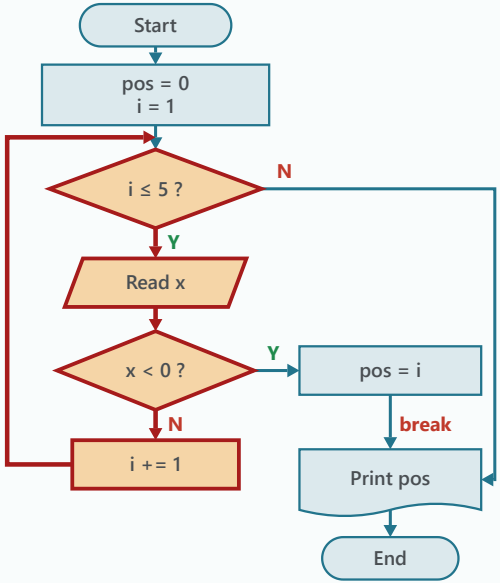
```
1 int pos = 0;
▶ for (int i = 1; i <= 5; i++) {
▶   int x = scanner.nextInt();
▶   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

VARIABLES

`pos = 0` `i = 3`

OUTPUT

```
4
9
```



x = 9: N; i = 3: Y

`x` = 9 is again non-negative; `i` = 3, 3 ≤ 5 → Yes.

Early Exit: Code Trace

```
1 int pos = 0;
2 for (int i = 1; i <= 5; i++) {
▶   int x = scanner.nextInt();
▶   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

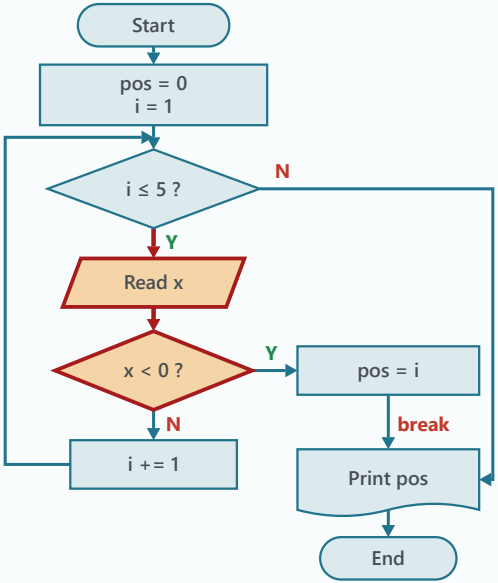
VARIABLES

`pos = 0` `i = 3`

`x = -2`

OUTPUT

```
4
9
-2
```



x = −2: −2 < 0 → Y

`x` = −2; −2 < 0 is **true** → Yes branch.

Early Exit: Code Trace

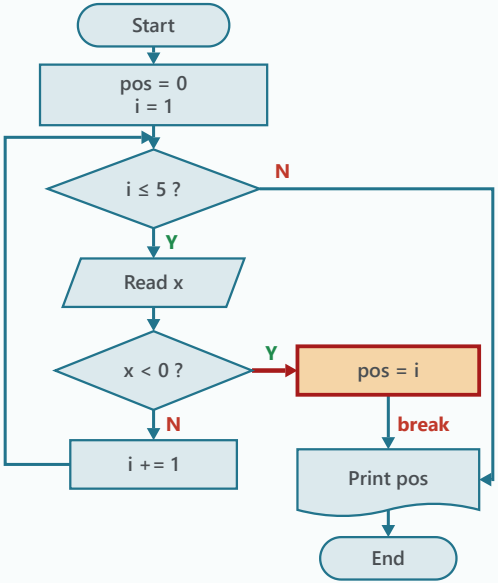
```
1 int pos = 0;
2 for (int i = 1; i <= 5; i++) {
3   int x = scanner.nextInt();
4   if (x < 0) {
▶     pos = i;
▶     break;
7   }
8 }
9 System.out.println("Position: " + pos);
```

VARIABLES

`pos = 3` `i = 3`

OUTPUT

```
4
9
-2
```



pos = 3

`pos` = 3. `break` : `i += 1` and the condition are skipped and the loop ends at once; 6 and 1 are never read.

Early Exit: Code Trace

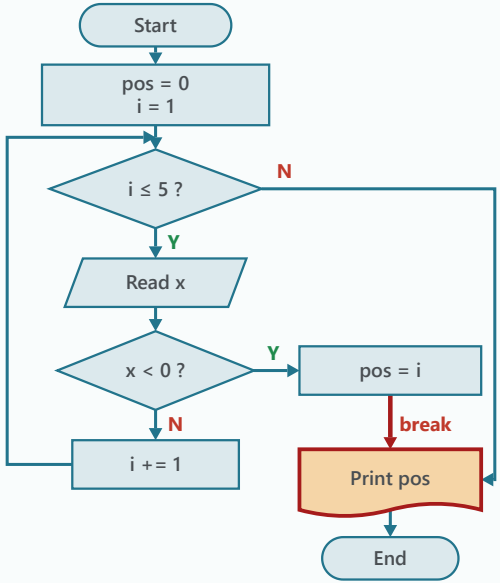
```
1 int pos = 0;
2 for (int i = 1; i <= 5; i++) {
3   int x = scanner.nextInt();
4   if (x < 0) {
5     pos = i;
6     break;
7   }
8 }
▶ System.out.println("Position: " + pos);
```

VARIABLES

`pos = 3`

OUTPUT

```
4
9
-2
Position: 3
```



Position: 3

The result is printed. With no negative input, the loop would exit through No at `i` = 6 and print `Position: 0`.

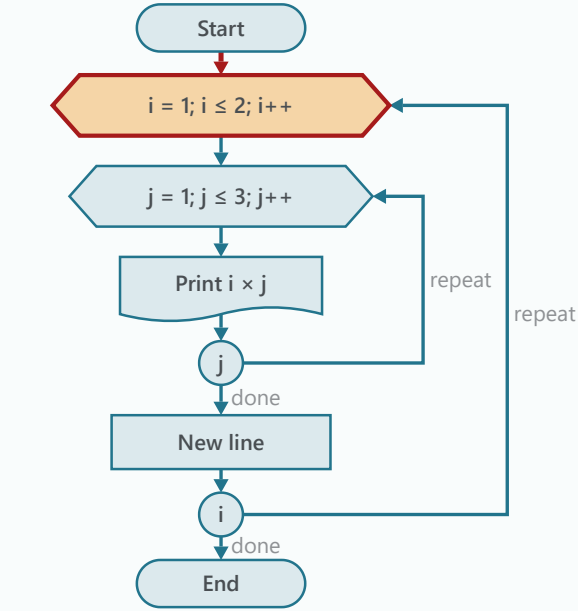
Nested Loops: Code Trace

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

VARIABLES

`i = 1`

OUTPUT



i = 1

The outer loop starts: `i` = 1; 1 ≤ 2 is true.

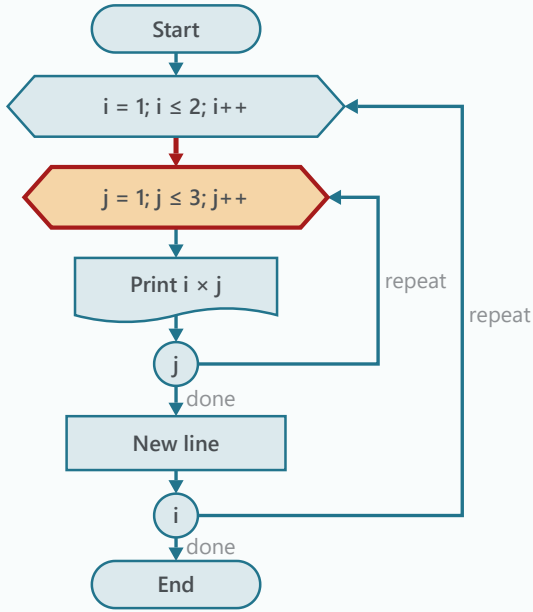
Nested Loops: Code Trace

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

VARIABLES

`i = 1` `j = 1`

OUTPUT

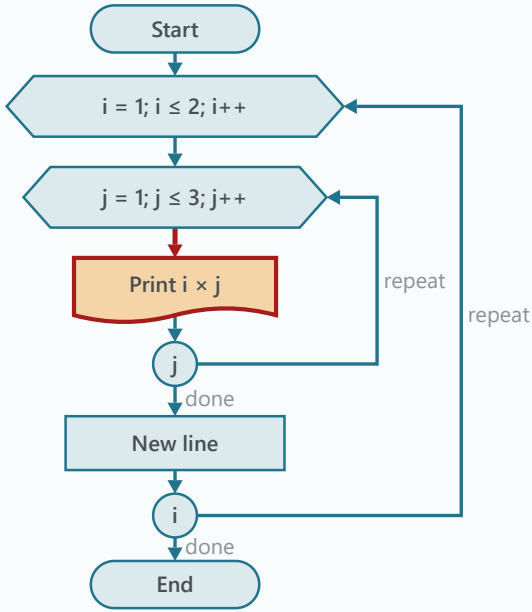


j = 1

The inner loop starts: `j` = 1.

Nested Loops: Code Trace

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



VARIABLES

`i = 1` `j = 1`

OUTPUT

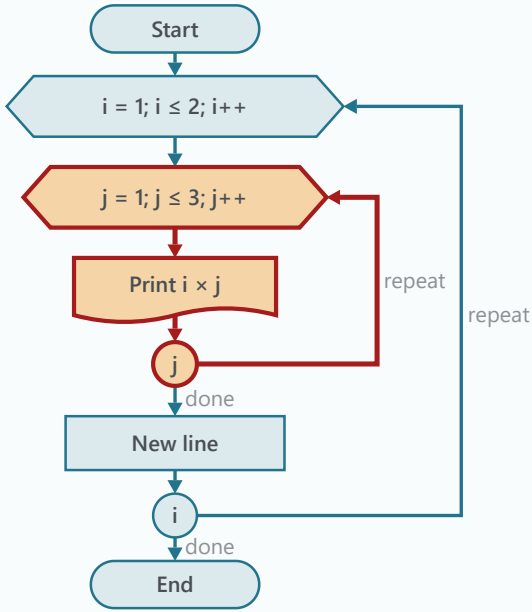
```
1
```

1 × 1 = 1

1 × 1 = 1 is printed.

Nested Loops: Code Trace

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



VARIABLES

`i = 1` `j = 2`

OUTPUT

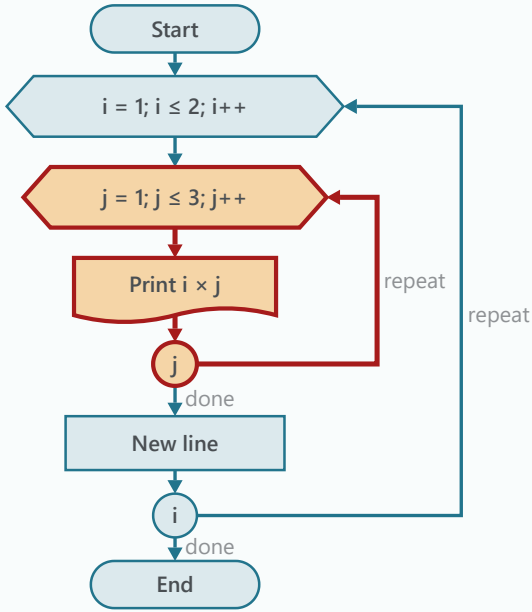
```
1 2
```

j = 2: 1 × 2 = 2

End of loop **j**: the repeat arrow leads back to the inner loop. `j` = 2, prints 2.

Nested Loops: Code Trace

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



VARIABLES

`i = 1` `j = 3`

OUTPUT

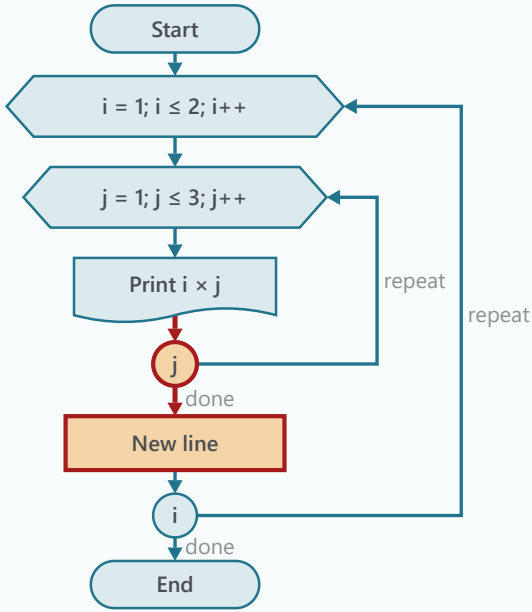
```
1 2 3
```

j = 3: 1 × 3 = 3

`j` = 3, prints 3.

Nested Loops: Code Trace

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



VARIABLES

`i = 1`

OUTPUT

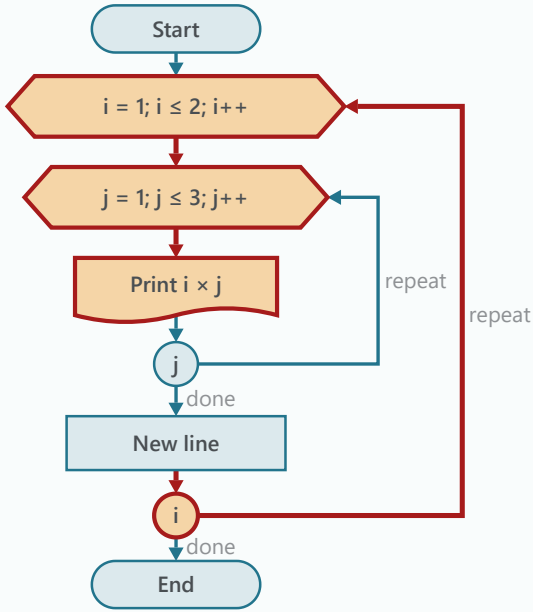
```
1 2 3
```

j = 4: 4 ≤ 3 → done

`j` = 4: 4 ≤ 3 is false, the inner loop ends (the done arrow). `println` starts a new line.

Nested Loops: Code Trace

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



i = 2, j = 1: 2

VARIABLES

`i = 2` `j = 1`

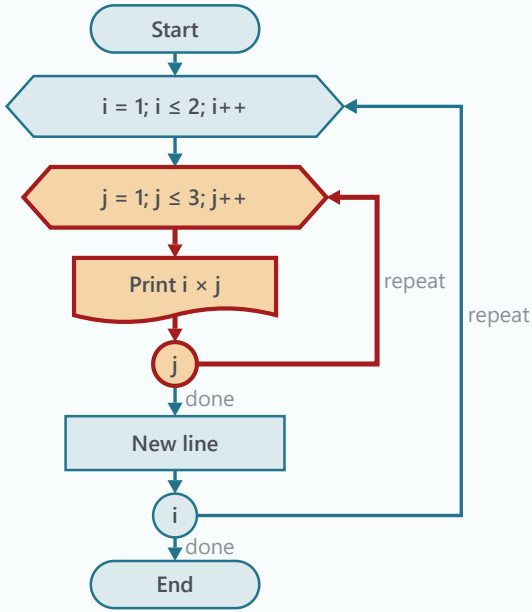
OUTPUT

```
1 2 3
2
```

End of loop **i**: back to the outer loop, `i` = 2. The inner loop starts **again** from `j` = 1: prints 2.

Nested Loops: Code Trace

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



VARIABLES

`i = 2` `j = 3`

OUTPUT

```
1 2 3
2 4 6
```

j = 2, 3: 4 and 6

`j` = 2 and 3: prints 4 and 6.

Nested Loops: Code Trace

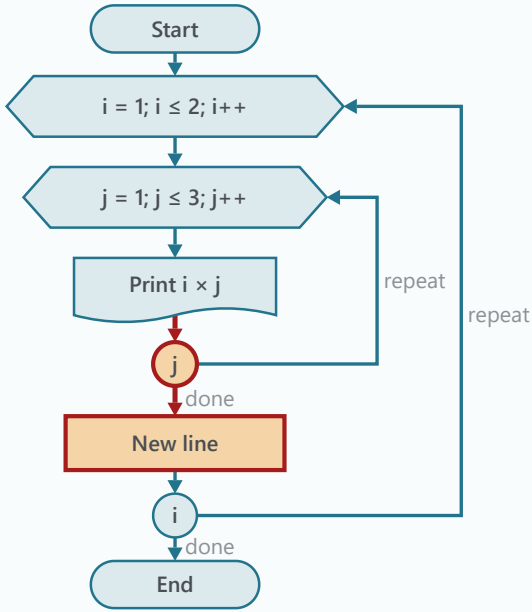
```
1 for (int i = 1; i <= 2; i++) {
▶   for (int j = 1; j <= 3; j++) {
3     System.out.print(i * j + " ");
4   }
▶   System.out.println();
6 }
```

VARIABLES

`i = 2`

OUTPUT

```
1 2 3
2 4 6
```

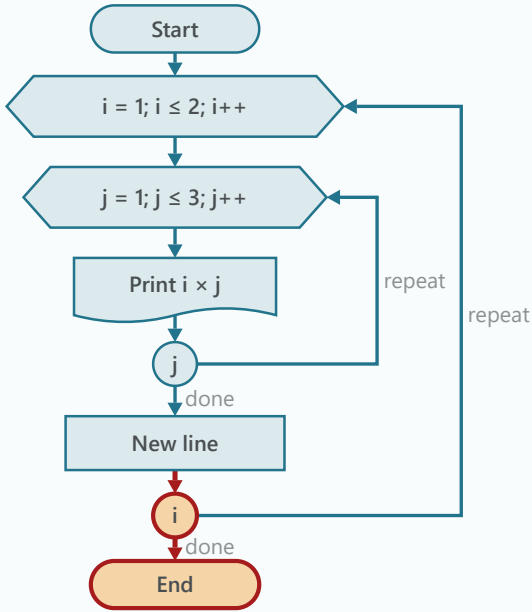


inner loop done

The inner loop ends again; new line.

Nested Loops: Code Trace

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



VARIABLES

`i = 3`

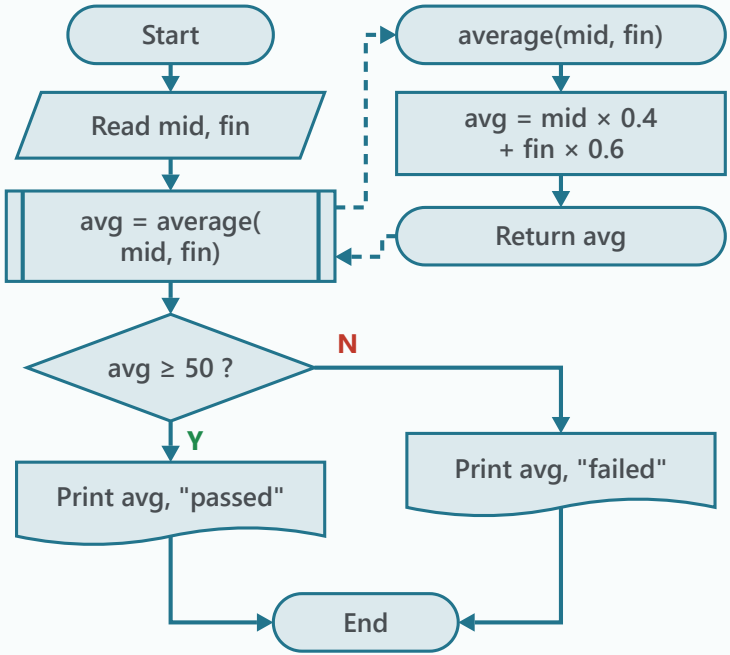
OUTPUT

```
1 2 3
2 4 6
```

i = 3: 3 ≤ 2 → done

`i` = 3: the outer loop ends. The inner body ran 2 × 3 = 6 times, `println` twice.

The Subprogram Symbol and Methods



- A long chart is split into meaningful parts: a part is drawn as a **separate chart** and called from the main chart with the **subprogram** symbol.
- The subprogram's Start box carries its name and inputs (**parameters**); it ends with a **Return** box.
- On a call the flow moves into the subprogram; at Return the result goes back to the caller and the flow continues right after the call (dashed arrows).
- In Java it is a **method**: `static double average(int mid, int fin)`, called as `average(mid, fin)`. `final` is a Java keyword, so the variable is named `fin`.
- The payoff: no redrawing the same steps, each part tested on its own, a short main chart.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
▶  public static void main(String[] args) {
▶    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
10   if (avg >= 50) System.out.println(avg + "
passed");
11   else System.out.println(avg + " failed");
12 }
```

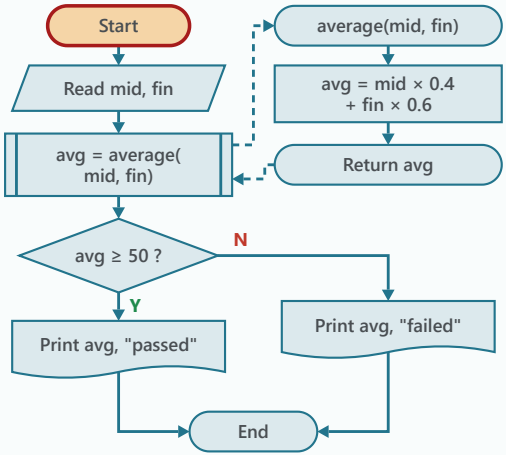
CALL STACK

`main()`

VARIABLES

—

OUTPUT



main starts

The program starts at `main` (Start).

Step 1 / 8

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
▶    int mid = scanner.nextInt();
▶    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
10   if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
12 }
```

CALL STACK

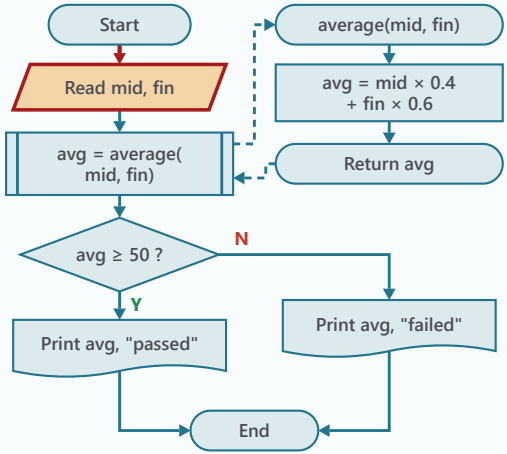
`main()`

VARIABLES

`mid = 40` `fin = 70`

OUTPUT

```
40 70
```



mid = 40, fin = 70

The user enters 40 and 70 on one line.

A Subprogram Call: Code Trace

```
▶ static double average(int mid, int fin) {
2   double avg = mid * 0.4 + fin * 0.6;
3   return avg;
4 }
5 public static void main(String[] args) {
6   Scanner scanner = new Scanner(System.in);
7   int mid = scanner.nextInt();
8   int fin = scanner.nextInt();
▶   double avg = average(mid, fin);
10  if (avg >= 50) System.out.println(avg + "
    passed");
11  else System.out.println(avg + " failed");
12 }
```

CALL STACK

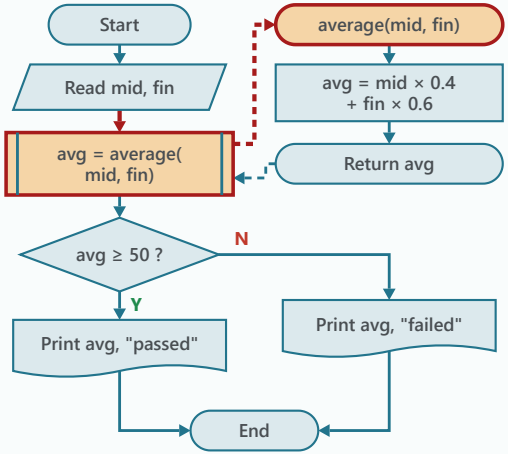
- `average(40, 70)`
- `main()`

VARIABLES

`mid = 40` `fin = 70`

OUTPUT

```
40 70
```



average(40, 70) called

Subprogram box: `average` is called; 40 and 70 are **copied** into the parameters.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
10   if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
12 }
```

CALL STACK

`average(40, 70)`

`main()`

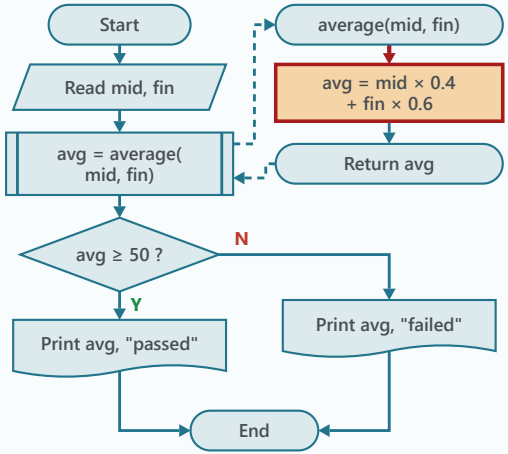
VARIABLES

`mid = 40` `fin = 70`

`avg = 58.0`

OUTPUT

```
40 70
```



avg = 16 + 42 = 58.0

The subprogram's own chart runs: 16 + 42 = 58.0. This `avg` is a local variable, separate from the `avg` in main.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
▶    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
▶    double avg = average(mid, fin);
10   if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
12 }
```

CALL STACK

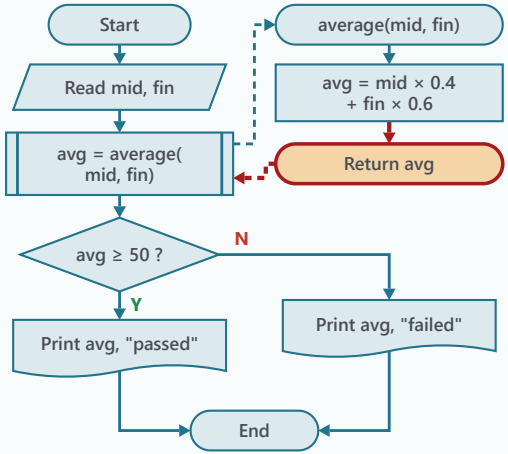
`main()`

VARIABLES

`mid = 40` `fin = 70` `avg = 58.0`

OUTPUT

```
40 70
```



Return 58.0

Return: the value goes back to the caller; `average` 's frame is popped off the stack.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
▶    if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
12 }
```

CALL STACK

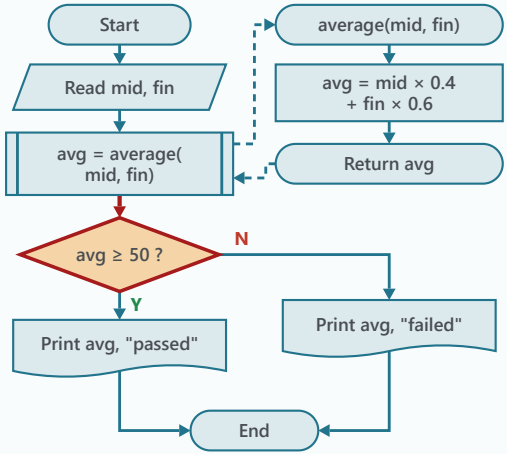
`main()`

VARIABLES

`mid = 40` `fin = 70` `avg = 58.0`

OUTPUT

```
40 70
```



58.0 ≥ 50 → Y

The flow continues after the call: 58.0 ≥ 50 is **true** → Yes.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
▶    if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
12 }
```

CALL STACK

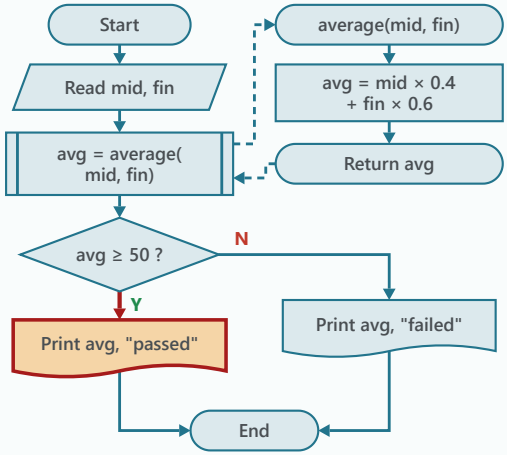
`main()`

VARIABLES

`mid = 40` `fin = 70` `avg = 58.0`

OUTPUT

```
40 70
58.0 passed
```



58.0 passed

The Yes branch prints.

A Subprogram Call: Code Trace

```
1  static double average(int mid, int fin) {
2    double avg = mid * 0.4 + fin * 0.6;
3    return avg;
4  }
5  public static void main(String[] args) {
6    Scanner scanner = new Scanner(System.in);
7    int mid = scanner.nextInt();
8    int fin = scanner.nextInt();
9    double avg = average(mid, fin);
10   if (avg >= 50) System.out.println(avg + "
   passed");
11   else System.out.println(avg + " failed");
▶  }
```

CALL STACK

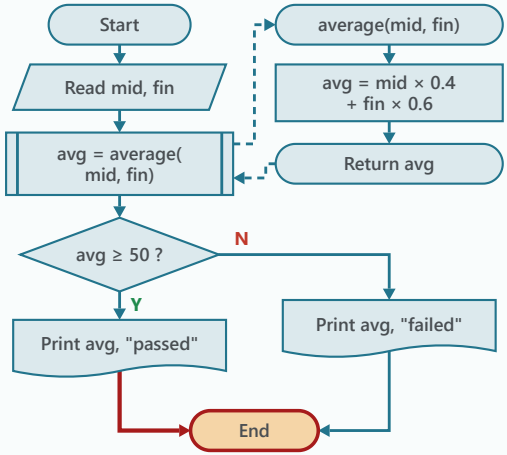
empty

VARIABLES

`mid = 40` `fin = 70` `avg = 58.0`

OUTPUT

```
40 70
58.0 passed
```

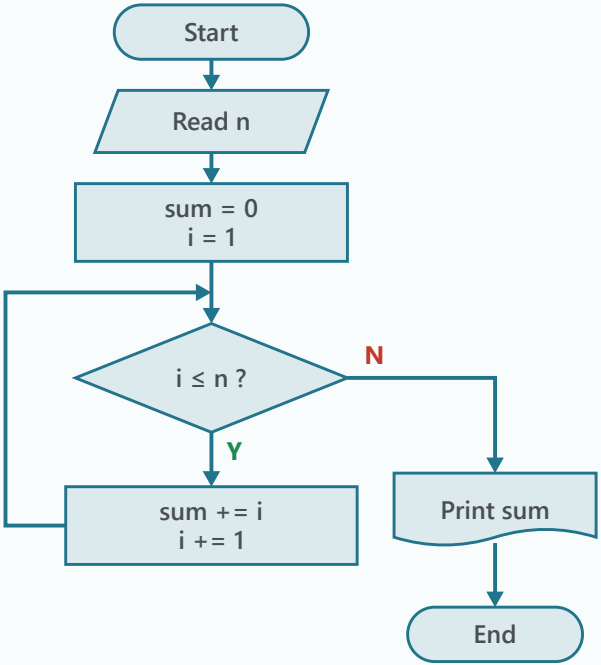


End

End. The same subprogram can be called again with other inputs: for 45 and 30, `36.0 failed`.

PRACTICE

Practice: The Sum 1 to n



Sequence (reading, initial values), iteration (adding) and sequence again (printing) together. `sum` is an **accumulator**: it starts at 0 and grows each round.

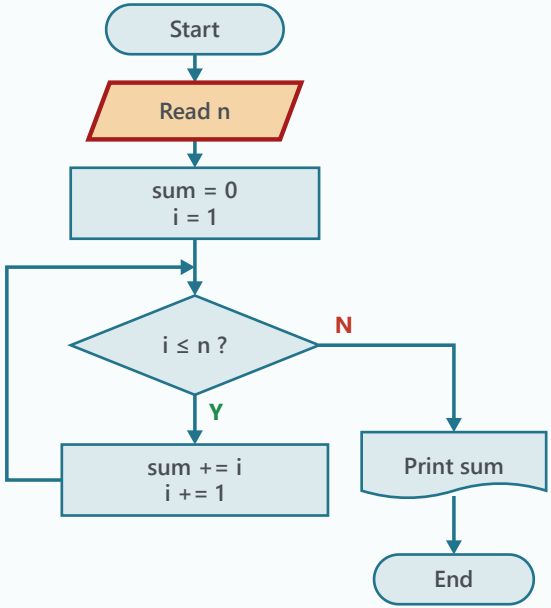
```
System.out.print("n: ");
int n = scanner.nextInt();
int sum = 0;
int i = 1;
while (i <= n) {
    sum += i;
    i += 1;
}
System.out.println("Sum: " + sum);
```

i	1	2	3	4	5	6
sum	1	3	6	10	15	15 (exit)

For n = 5 the output is `Sum: 15` (1 + 2 + 3 + 4 + 5).

The Sum 1 to n: Code Trace

```
System.out.print("n: ");
int n = scanner.nextInt();
int sum = 0;
int i = 1;
while (i <= n) {
    sum += i;
    i += 1;
}
System.out.println("Sum: " + sum);
```



n = 4

VARIABLES

`n = 4`

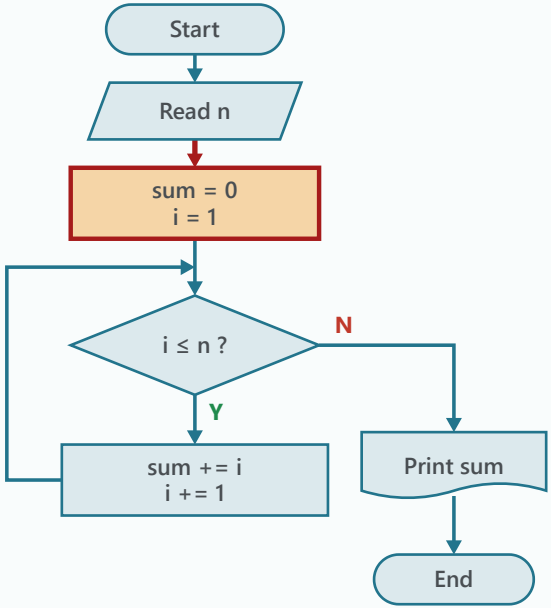
OUTPUT

```
n: 4
```

The prompt prints and the user enters 4: `n` = 4.

The Sum 1 to n: Code Trace

```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
▶ int sum = 0;
▶ int i = 1;
5 while (i <= n) {
6     sum += i;
7     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```



VARIABLES

`n = 4` `sum = 0` `i = 1`

OUTPUT

```
n: 4
```

sum = 0, i = 1

Initial values: accumulator `sum` = 0, counter `i` = 1.

The Sum 1 to n: Code Trace

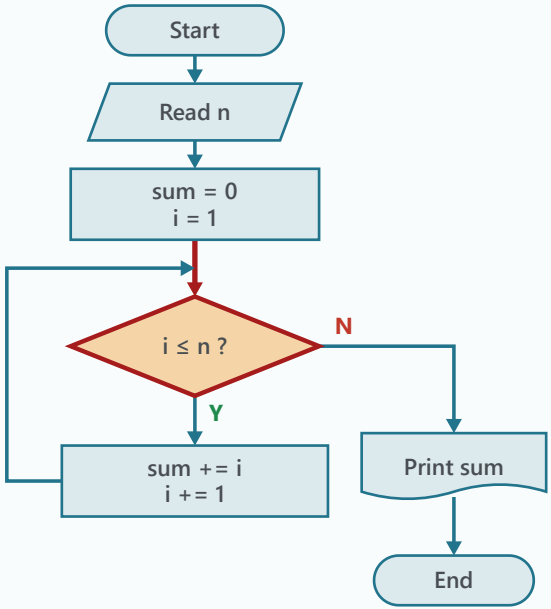
```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
▶ while (i <= n) {
6     sum += i;
7     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```

VARIABLES

`n = 4` `sum = 0` `i = 1`

OUTPUT

```
n: 4
```



1 ≤ 4 → Y

Condition: 1 ≤ 4 is **true** → Yes, enter the body.

The Sum 1 to n: Code Trace

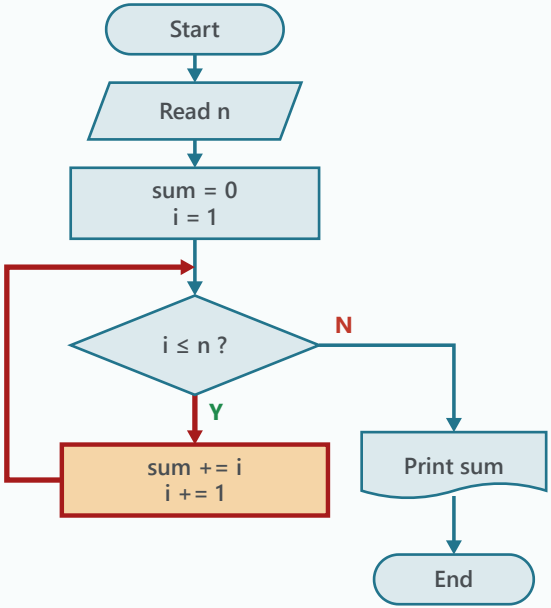
```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
5 while (i <= n) {
▶     sum += i;
▶     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```

VARIABLES

`n = 4` `sum = 1` `i = 2`

OUTPUT

```
n: 4
```



sum = 0 + 1 = 1, i = 2

`sum` = 0 + 1 = 1, `i` = 2; back to the decision.

The Sum 1 to n: Code Trace

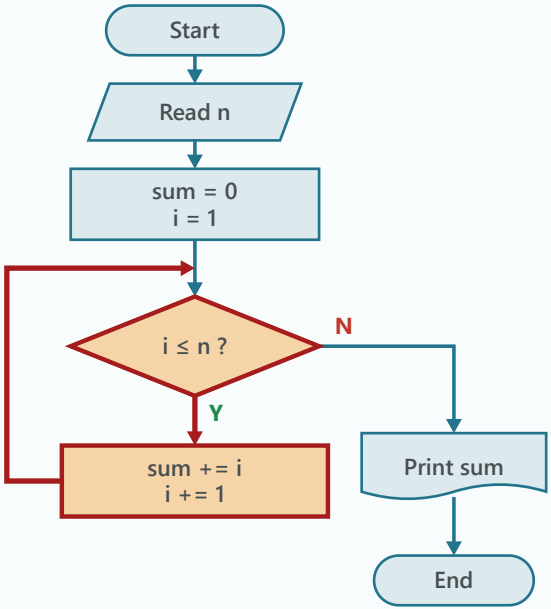
```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
▶ while (i <= n) {
▶     sum += i;
▶     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```

VARIABLES

`n = 4` `sum = 3` `i = 3`

OUTPUT

```
n: 4
```



2 ≤ 4 → sum = 3, i = 3

2 ≤ 4 is true: `sum` = 1 + 2 = 3, `i` = 3.

The Sum 1 to n: Code Trace

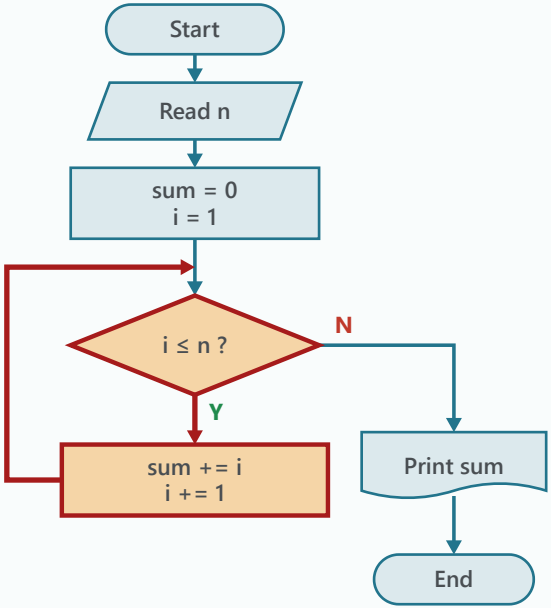
```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
▶ while (i <= n) {
▶     sum += i;
▶     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```

VARIABLES

`n = 4` `sum = 6` `i = 4`

OUTPUT

```
n: 4
```

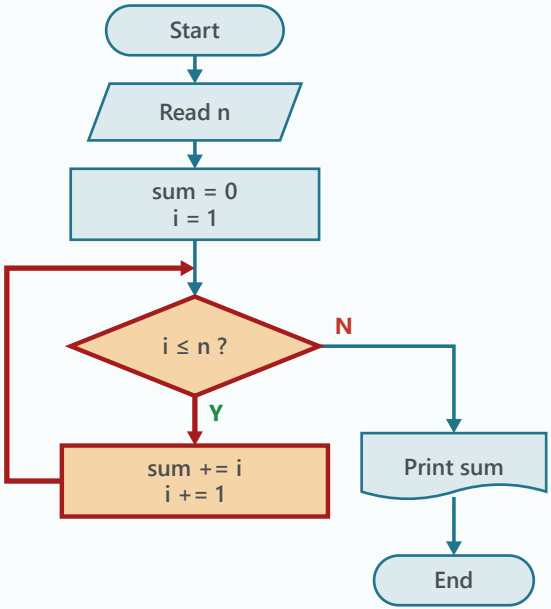


3 ≤ 4 → sum = 6, i = 4

3 ≤ 4 is true: `sum` = 3 + 3 = 6, `i` = 4.

The Sum 1 to n: Code Trace

```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
▶ while (i <= n) {
▶     sum += i;
▶     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```



VARIABLES

`n = 4` `sum = 10` `i = 5`

OUTPUT

```
n: 4
```

4 ≤ 4 → sum = 10, i = 5

4 ≤ 4 is true: `sum` = 6 + 4 = 10, `i` = 5.

The Sum 1 to n: Code Trace

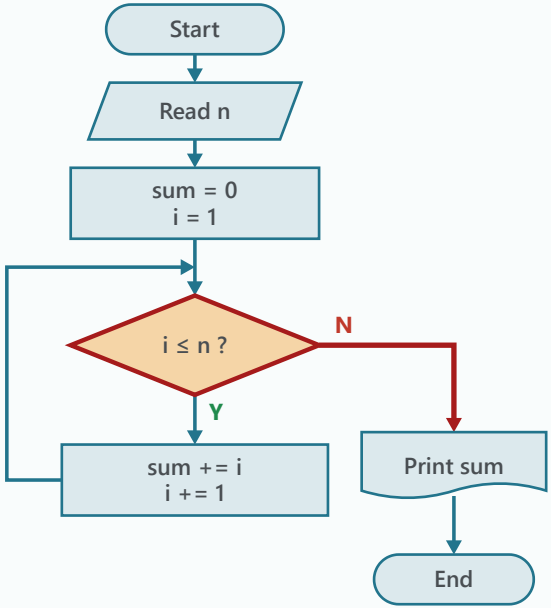
```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
▶ while (i <= n) {
6     sum += i;
7     i += 1;
8 }
9 System.out.println("Sum: " + sum);
```

VARIABLES

`n = 4` `sum = 10` `i = 5`

OUTPUT

```
n: 4
```

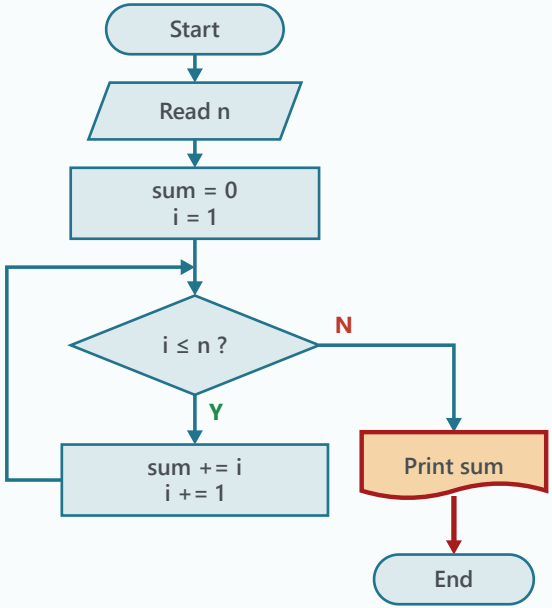


5 ≤ 4 → N

Condition: 5 ≤ 4 is **false** → the No exit, leave the loop.

The Sum 1 to n: Code Trace

```
1 System.out.print("n: ");
2 int n = scanner.nextInt();
3 int sum = 0;
4 int i = 1;
5 while (i <= n) {
6     sum += i;
7     i += 1;
8 }
▶ System.out.println("Sum: " + sum);
```



VARIABLES

`n = 4` `sum = 10` `i = 5`

OUTPUT

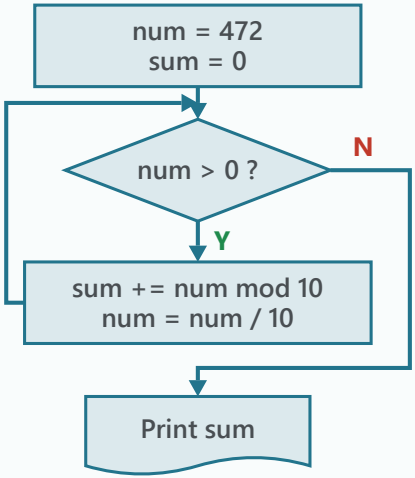
```
n: 4
Sum: 10
```

output: Sum: 10

The result is printed: 1 + 2 + 3 + 4 = 10. The body ran 4 times, the condition 5 times.

PRACTICE

Desk Checking: The Trace Table



Step	num mod 10	sum	num	num > 0 ?
start	–	0	472	Y
round 1	2	2	47	Y
round 2	7	9	4	Y
round 3	4	13	0	N → Print 13

- **Desk check**: trace the chart or code on paper with one input, no computer.
- **Trace table**: one column per variable, one row per round. The condition column shows how often the loop ran: 3 rounds, 4 tests.
- `num / 10` is integer division: 47 / 10 = 4. Digits are taken right to left.
- The table exposes errors early: forget the `num = num / 10` box and the column stays 472 forever, an infinite loop.

```
int num = 472, sum = 0;
while (num > 0) {
  sum += num % 10;
  num /= 10;
}
System.out.println(sum);
```

PRACTICE

Testing a Flowchart: Test Cases

Positive / negative / zero chart

Input	Why this one?	Expected
5	Yes path of decision 1	positive
−2	Yes path of decision 2	negative
0	No path of both decisions	zero
1 and −1	either side of the boundary	positive, negative

Sum 1 to n chart

n	Why this one?	Expected
0	body never runs	Sum: 0
1	one round	Sum: 1
4	typical case	Sum: 10
−3	invalid input	Sum: 0 (is that what we want?)

- One input is not enough: **every exit** of every decision box should run at least once (**path coverage**).
- **Boundary values** sit on either side of where the condition flips: 0 and 1 for `num > 0`; 49 and 50 for `grade ≥ 50`. Off-by-one errors show up here.
- For loops, three cases: inputs that make it run **zero** times, **once** and **many** times.
- Work out the expected output by hand before running, then compare. Any difference means an error in the chart, the code or the expectation.

PRACTICE

Ways to Express an Algorithm

Notation	Odd/even example	Strength	Weakness
Plain language	"Read the number; if dividing by 2 leaves 0, print even, otherwise odd."	Anyone can read it	Can be ambiguous
Flowchart	Read → Process → Decision → Print even / Print odd	Shows the flow visually; branches and loops are easy to see	Unwieldy for large problems
Pseudocode	`READ n; IF n mod 2 = 0 THEN PRINT "even" ELSE PRINT "odd"`	Quick to write, close to code	No standard syntax
Java	`if (num % 2 == 0) … else …`	A computer can run it	Must follow strict syntax

In software engineering the modern counterpart of the flowchart is the **UML activity diagram**: same idea, with a filled circle for the start, a diamond for decisions and rounded boxes for activities.

PRACTICE

Tools for Drawing Flowcharts

Tool	What it does	Note
Pen and paper	The fastest sketch; the only option in an exam	Tedious to correct
draw.io (diagrams.net)	Free in-browser drawing tool with ready-made flowchart symbols	Draws, does not run
Microsoft Visio	Commercial diagramming program with standard symbol libraries	Paid
Flowgorithm	**Runs** the chart and tracks variables; generates code in many languages, Java included	Free, for teaching
Mermaid	Draws charts from text; GitHub and many Markdown tools render it directly	Lives alongside the code

The odd/even chart in Mermaid

```
flowchart TD
  A([Start]) --> B[/Read num/]
  B --> C{num mod 2 = 0 ?}
  C -- Yes --> D[Print even]
  C -- No --> E[Print odd]
  D --> F([End])
  E --> F
```

- `([ ])` Start/End, `[/ /]` input, `{ }` decision, `[ ]` process; `-->` a flow line, `-- Yes -->` a labeled exit.
- `TD`: top-down layout; the tool places the boxes itself.
- Whatever the tool, the rules are the same: one Start, labeled decision exits, no crossing arrows.

Summary

Topic	Key points
Flowchart	an algorithm drawn with symbols and arrows; a language-independent plan
Symbols	oval Start/End · parallelogram input · rectangle process · diamond decision · hexagon loop · wavy box print · circle connector
Rules	one Start; top to bottom; two labeled exits (Y/N) on decisions; no crossing lines
Sequence	steps one after another, each once; dependent steps must stay in order
Selection	question K → Yes: P, No: Q; Java `if/else`; nested decisions for three or more paths
Iteration	back arrow + decision; initialization, condition, update; Java `while`, `for`, `do-while`
Patterns	counter, accumulator, flag; sentinel loop; early exit (`break`); subprogram = method
Pitfalls	`<` / `≤` boundary errors; forgetting the update → infinite loop

SUMMARY

Common Mistakes

A decision with one exit

Two arrows must leave a decision, labeled Y/N; otherwise what happens on "no" is undefined.

Wrong symbol

A rectangle for keyboard input, a parallelogram for a calculation. Input = parallelogram, process = rectangle.

Using = to compare

`if (result = 0)` does not compile: `error: incompatible types: int cannot be converted to boolean`. Write `result == 0`.

Forgetting the initial value

Without the "count = 1" box, Java complains too: `error: variable count might not have been initialized`.

Forgetting the update

Without the "count += 1" box the condition stays true: an **infinite loop**. The compiler cannot catch this.

Boundary error

Writing `count < 10` for "1 through 10" skips 10. Test the condition with the last value.

A semicolon after the loop header

`while (count <= 10);` compiles, but the loop body is empty; the counter never grows and the program hangs.

Back arrow to the wrong place

If the back arrow returns **above** the "count = 1" box, the counter resets to 1 every round: another infinite loop. It must return to the decision box.

SUMMARY

Sample Problems

1. Which symbols draw the steps "get the user's age", "age ≥ 18?", "tax = price × 0.2" and "print the result"?
2. Chart: sum = 0, i = 1; while i ≤ 4, sum += i and i += 1; then Print sum. What is the output? What is i on exit?
3. What does the odd/even chart print for **−3**? (What is `-3 % 2` in Java?)
4. In the 1-to-10 chart, how many numbers would print with the condition `count < 10`? How many times would the decision be tested?
5. Starting from num = 1, how many times does the loop "while num < 100, num = num × 2" run, and what is num at the end?

▶ **Show answers**

SUMMARY

Exercises

1. Draw a flowchart that reads a rectangle's sides and prints its area and perimeter (sequence).
2. Draw a chart that finds the largest of three numbers. Trace it for 3, 9, 5.
3. Draw a chart for the sum of the **even** numbers from 1 to n. What is it for n = 10?
4. Draw a chart for n! (factorial) and translate it to Java. What is it for n = 5?
5. Draw a chart that reads numbers until the user enters 0, then prints the sum. Input: 4 7 2 0.
6. Draw a chart that turns a 0–100 score into a letter grade (≥ 85 A, ≥ 70 B, ≥ 50 C, else F).

▶ **Answers / hints**

References

- ISO 5807:1985, *Information processing — Documentation symbols and conventions for data, program and system flowcharts*.
- C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM* 9(5), 1966.
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson, 2017 (Chapters 4–5: control statements and activity diagrams).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019.
- Oracle, *The Java Tutorials*: "Control Flow Statements" (docs.oracle.com/javase/tutorial).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly, 2022.

The End

Chapter 1.4: Flowcharts

Previous: Chapter 1.3: Algorithms · Next: Chapter 2.1: Programming