# **Private methods in interfaces**

**📅 When Were Private Methods in Interfaces Introduced?**

* **Introduced in Java 9** (released in September 2017).

**❓ Why Were Private Methods Introduced in Interfaces?**

Before Java 9:

* Interfaces could have **default** and **static** methods (since Java 8).
* But there was **no way to share common code** between these methods **within** the interface.

👉 Java 9 added **private** and **private static** methods in interfaces to **reduce code duplication** and **improve encapsulation**.

**✅ When to Use Private Methods in Interfaces**

Use **private methods** in interfaces when:

* You want to **reuse logic** between multiple default or static methods.
* You want to **hide helper logic** within the interface (just like private methods in classes).
* You aim for **cleaner, more modular interface code** without exposing unnecessary logic to implementing classes.

**📌 Types of Private Methods Allowed in Interfaces**

1. private – instance-level helper for default methods.
2. private static – static-level helper for static methods.

**🌍 Real-World Example: Logging in Utility Interface**

interface Logger {

// Public static method accessible externally

static void logInfo(String message) {

log("INFO", message);

}

// Public default method usable by implementing classes

default void logWarning(String message) {

log("WARNING", message);

}

// Private helper method to avoid code duplication

private static void log(String level, String message) {

System.out.println("[" + level + "] " + message);

}

}

**🔍 Usage in a Class:**

public class AppLogger implements Logger {

public void customLog() {

logWarning("Low memory");

}

public static void main(String[] args) {

Logger.logInfo("System started"); // static method

new AppLogger().customLog(); // default method

}

}

**🧾 Output:**

[INFO] System started

[WARNING] Low memory

Here, the private method log() is reused by both the static and default methods, but it's hidden from external access.

**🧠 Benefits Recap**

* Promotes **code reuse** inside interfaces.
* Keeps **helper logic encapsulated**.
* Keeps the interface **clean and readable**.

# **🧰 1. Java Collectors API**

**✅ What is it?**

The Collectors class is part of the **Java Streams API** (added in **Java 8**) and provides **reduction operations** like collecting elements into a list, set, map, or string.

It's part of java.util.stream.Collectors

**🔄 Common Use Cases:**

* Collect elements from a stream into a collection
* Group or partition data
* Summarize numeric data (sum, avg, etc.)
* Join strings

**📌 Common Collectors Methods:**

| **Method** | **Purpose** |
| --- | --- |
| Collectors.toList() | Collect elements into a List |
| Collectors.toSet() | Collect into a Set |
| Collectors.toMap() | Collect into a Map |
| Collectors.joining() | Concatenate strings |
| Collectors.groupingBy() | Group data by key |
| Collectors.partitioningBy() | Partition data by boolean condition |
| Collectors.counting() | Count elements |

**🌍 Real-World Example: List of Employees by Department**

class Employee {

String name;

String department;

Employee(String name, String department) {

this.name = name;

this.department = department;

}

public String getDepartment() { return department; }

public String getName() { return name; }

}

public class CollectorDemo {

public static void main(String[] args) {

List<Employee> employees = List.of(

new Employee("Alice", "HR"),

new Employee("Bob", "IT"),

new Employee("Charlie", "HR")

);

// Grouping by department

Map<String, List<Employee>> empByDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

empByDept.forEach((dept, empList) -> {

System.out.println(dept + ": " + empList.stream().map(Employee::getName).toList());

});

}

}

**🔒 2. Immutable Collections**

**✅ What Are They?**

Immutable collections are **read-only collections** — once created, they **cannot be modified** (no add/remove/update).

Introduced in Java 9 via factory methods in the List, Set, and Map interfaces.

**🛠️ Creation Syntax:**

List<String> names = List.of("Alice", "Bob");

Set<Integer> ids = Set.of(101, 102, 103);

Map<String, Integer> scores = Map.of("Math", 90, "English", 85);

**⚠️ Rules / Constraints:**

* **No nulls allowed** — will throw NullPointerException
* **UnsupportedOperationException** is thrown if modified
* **Fixed size and content** — fully immutable
* Duplicate keys in Map.of(...) will throw IllegalArgumentException

**🌍 Real-World Use Case: Country Codes**

public class CountryCodes {

public static void main(String[] args) {

Map<String, String> countries = Map.of(

"IN", "India",

"US", "United States",

"UK", "United Kingdom"

);

System.out.println(countries.get("IN")); // Output: India

// countries.put("AU", "Australia"); // ❌ Throws UnsupportedOperationException

}

}

**🧠 When to Use Them**

**🔄 Collectors API**

* When working with **Streams** and you need to **collect results** efficiently.
* Great for **data transformation**, **filtering**, **grouping**, and **aggregations**.
* Use in **report generation**, **data processing**, **ETL-like flows**.

**🔒 Immutable Collections**

* When you want **thread safety without synchronization**.
* When returning **read-only views** from APIs.
* Great for **config values**, **constants**, **predefined lists**.

**✅ Summary**

| **Feature** | **Use When** | **Benefit** |
| --- | --- | --- |
| **Collectors API** | You’re streaming data and need to collect, group, or transform it | Cleaner, functional code |
| **Immutable Collections** | You want read-only, safe-to-share collections | Safer and no accidental mutation |

**❓ MCQs on Collectors API & Immutable Collections**

**Q1. What is the purpose of Collectors.toMap()?**  
A) Convert a map into a list  
B) Collect stream elements into a map  
C) Count elements in a stream  
D) None of the above

**✅ Answer:** B

**Q2. What does Collectors.groupingBy() return?**  
A) A stream  
B) A list  
C) A map where the key is the group  
D) A set

**✅ Answer:** C

**Q3. Which method creates an immutable list in Java 9+?**  
A) new ArrayList<>()  
B) Collections.unmodifiableList()  
C) List.of(...)  
D) Arrays.asList()

**✅ Answer:** C

**Q4. Which of the following will throw an exception when using List.of()?**  
A) Adding null to the list  
B) Adding duplicate values  
C) Reading from the list  
D) Printing the list

**✅ Answer:** A

**Q5. What is the result of this code?**

Map<String, String> map = Map.of("a", "Apple", "b", "Banana", "c", "Cherry");

map.put("d", "Date");

A) Adds "d" successfully  
B) Throws NullPointerException  
C) Throws UnsupportedOperationException  
D) Compilation error

**✅ Answer:** C

**📅 When Was var Introduced in Java?**

* Introduced in **Java 10** (March 2018).
* It's part of **local variable type inference**.

**❓ Why Was var Introduced?**

To:

* **Reduce verbosity** (especially with generics and long type names).
* Improve **code readability** when the type is obvious.
* Align Java with modern languages like Kotlin, Scala, TypeScript, which support type inference.

**✅ What var Does**

* Lets the **compiler infer the type** of a **local variable** based on the value it’s initialized with.
* You **must assign a value** at declaration time.

**📌 Syntax**

var name = "Alice"; // Compiler infers it as String

var age = 30; // Inferred as int

**⚠️ Rules of var**

| **Rule** | **Description** |
| --- | --- |
| ✅ Only for **local variables** | Not for class fields, method parameters, or return types. |
| ✅ Must have an initializer | You **cannot** do var x; |
| ✅ Type is inferred at **compile time** | It’s still strongly typed. |
| ❌ Not allowed with null | var data = null; → ❌ |

**🚫 Not Allowed**

var x; // ❌ Error: Cannot use 'var' without initializer

var y = null; // ❌ Error: Cannot infer type from null

**🧠 When to Use var**

Use var when:

* The type is **obvious** from the right-hand side.
* You want to **avoid redundant types**.
* You're dealing with **generic or complex types**.

Avoid var when:

* It makes code **less readable or confusing**.
* You're trying to **look clever** — clarity is more important!

**🌍 Real-World Examples**

**🔸 Example 1: Simple Types**

var name = "John"; // String

var age = 25; // int

var isActive = true; // boolean

**🔸 Example 2: Collections**

var list = new ArrayList<String>();

list.add("Java");

list.add("Python");

for (var lang : list) {

System.out.println(lang);

}

**🔸 Example 3: Map with Streams**

var map = Map.of("IN", "India", "US", "United States");

for (var entry : map.entrySet()) {

System.out.println(entry.getKey() + " => " + entry.getValue());

}

**🔸 Example 4: Working with Streams**

var books = List.of("Java", "Python", "Kotlin");

var filtered = books.stream()

.filter(b -> b.startsWith("J"))

.collect(Collectors.toList());

System.out.println(filtered);

**✨ Benefits Summary**

| **Benefit** | **Description** |
| --- | --- |
| Cleaner Code | Avoids repetitive type declarations |
| Better with Generics | Easier to work with complex types |
| Safer than dynamic typing | Type is still enforced by compiler |

**🧠 MCQs: var Keyword in Java**

**Q1. When was the var keyword introduced in Java?**

A) Java 8  
B) Java 9  
C) Java 10  
D) Java 11

**✅ Answer:** C) Java 10

**Q2. What is the purpose of var in Java?**

A) It allows dynamic typing like JavaScript  
B) It lets the compiler infer the variable’s type  
C) It is used to declare constant variables  
D) It removes the need for import statements

**✅ Answer:** B) It lets the compiler infer the variable’s type

**Q3. Which of the following is a valid use of var?**

A) var name;  
B) var age = 25;  
C) var x = null;  
D) public var salary = 5000;

**✅ Answer:** B) var age = 25;

**Q4. Can var be used for method parameters or return types?**

A) Yes, always  
B) Only in interfaces  
C) No  
D) Only in lambda expressions

**✅ Answer:** C) No

**Q5. What will be the type of the variable var list = new ArrayList<String>();?**

A) Object  
B) List  
C) ArrayList  
D) String

**✅ Answer:** C) ArrayList

**Q6. What will happen if you write var x = null;?**

A) The code runs fine  
B) x is of type Object  
C) Compilation error  
D) x becomes dynamic

**✅ Answer:** C) Compilation error

**Q7. Choose the best practice when using var:**

A) Use var everywhere to write less code  
B) Use var when the type is obvious or reduces clutter  
C) Use var to avoid learning data types  
D) Avoid var completely for safety

**✅ Answer:** B) Use var when the type is obvious or reduces clutter

**Q8. Is var a reserved keyword in Java?**

A) Yes, and it can’t be used as a variable name  
B) No, it’s just a contextual keyword  
C) It can be used as a method name  
D) Only reserved in Java 11+

**✅ Answer:** B) No, it’s just a contextual keyword

**🔁 Switch Expressions in Java**

**🗓️ When Were They Introduced?**

* **Preview**: Java 12
* **Standardized**: Java 14 (March 2020)
* Syntax enhancement to traditional switch statement

**❓ Why Were They Introduced?**

Traditional switch had several issues:

* Verbose and repetitive
* Fall-through behavior (unless using break)
* Error-prone (e.g., forgetting break)
* No way to return a value directly

🆕 **Switch expressions** fix these by:

* Allowing return values
* Removing fall-through by default
* Using arrow (->) syntax for clarity

**✅ New Syntax**

String dayType = switch (day) {

case MONDAY, FRIDAY -> "Workday";

case SATURDAY, SUNDAY -> "Weekend";

default -> "Midweek";

};

✨ Notice: The result of the switch can be directly assigned to a variable.

**⚖️ Traditional vs Modern**

**Old Style:**

String result;

switch (day) {

case "MONDAY":

case "FRIDAY":

result = "Workday";

break;

default:

result = "Other";

}

**New Style:**

String result = switch (day) {

case "MONDAY", "FRIDAY" -> "Workday";

default -> "Other";

};

**🧠 When to Use Switch Expressions**

Use when:

* You need to assign a value based on a condition
* You want **cleaner, safer, and less verbose** code
* You want to **avoid fall-through bugs**
* You're mapping enums, strings, or primitive values

**🌍 Real-World Example**

**🎯 Example: Payment Mode**

enum PaymentMode { CASH, CARD, UPI }

public class PaymentSwitch {

public static void main(String[] args) {

PaymentMode mode = PaymentMode.CARD;

String status = switch (mode) {

case CASH -> "Pay with cash at counter";

case CARD -> "Swipe your card";

case UPI -> "Scan the QR code";

};

System.out.println("Instruction: " + status);

}

}

**📚 MCQs: Switch Expressions in Java**

**Q1. In which version were switch expressions standardized in Java?**

A) Java 11  
B) Java 12  
C) Java 13  
D) Java 14

**✅ Answer:** D) Java 14

**Q2. Which of the following is a benefit of switch expressions?**

A) Allows checked exceptions  
B) Enables method overloading  
C) Allows returning values from switch directly  
D) Automatically adds break after each case

**✅ Answer:** C) Allows returning values from switch directly

**Q3. What will the following code output?**

String lang = "Python";

String type = switch (lang) {

case "Java", "C++" -> "Compiled";

case "Python" -> "Interpreted";

default -> "Unknown";

};

System.out.println(type);

A) Compiled  
B) Interpreted  
C) Unknown  
D) Runtime Error

**✅ Answer:** B) Interpreted

**Q4. Which syntax is correct for a switch expression that returns a value?**

A) switch(day) { case MONDAY: return "Work"; }  
B) switch(day) -> { case MONDAY: "Work"; }  
C) String result = switch(day) { case MONDAY -> "Work"; };  
D) switch(day) { MONDAY -> "Work"; }

**✅ Answer:** C) String result = switch(day) { case MONDAY -> "Work"; };

**Q5. Can switch expressions be used with enums?**

A) Yes  
B) No  
C) Only in preview mode  
D) Only with Strings

**✅ Answer:** A) Yes

**🗓️ When Were Text Blocks Introduced in Java?**

* **Preview Feature**: Java **13** (Sept 2019)
* **Standardized**: Java **15** (Sept 2020)

**❓ Why Were Text Blocks Introduced?**

Before text blocks, writing multi-line strings in Java was:

* ❌ Verbose
* ❌ Hard to read
* ❌ Full of escape sequences (\n, \", \\)

**Text blocks** fix this by allowing:

✅ Multi-line strings  
✅ No need for \n  
✅ Preserves formatting  
✅ Easy to embed SQL, JSON, HTML, XML, etc.

**✅ Syntax**

String html = """

<html>

<body>

<h1>Hello, Java!</h1>

</body>

</html>

""";

* Uses triple double-quotes """
* Automatically handles newlines and indentation

**🧠 When to Use Text Blocks**

Use Text Blocks when:

* You're working with **multi-line content**
* Writing **SQL**, **HTML**, **JSON**, **XML**, **scripts**
* You want **cleaner, more readable** code

Avoid using them for:

* Simple one-line strings
* Strings that are dynamically constructed piece-by-piece

**🌍 Real-World Examples**

**🔸 Example 1: HTML Template**

String emailTemplate = """

<html>

<body>

<p>Dear Customer,</p>

<p>Your order has been shipped.</p>

</body>

</html>

""";

**🔸 Example 2: SQL Query**

String sql = """

SELECT id, name, email

FROM users

WHERE status = 'active'

ORDER BY name;

""";

**🔸 Example 3: JSON Response**

String json = """

{

"status": "success",

"data": {

"user": "john\_doe",

"role": "admin"

}

}

""";

**⚙️ Key Rules & Notes**

| **Rule** | **Description** |
| --- | --- |
| Starts with """ | Must be the only thing on the line after = |
| Line breaks | Automatically preserved |
| Escaping | Fewer escapes (\" only if embedded inside text) |
| Indentation | Automatically adjusted based on closing """ position |

**✨ Benefits Summary**

| **Benefit** | **What it means** |
| --- | --- |
| Less boilerplate | No need for \n, +, etc. |
| Better readability | Matches the output layout |
| Useful for templates | Ideal for HTML, SQL, JSON, XML |

**📅 When Were Records Introduced?**

* **Preview in Java 14** (March 2020)
* **Standardized in Java 16** (March 2021)

**❓ Why Were Records Added?**

1. **Boilerplate reduction**
   * Traditional data carrier (POJO[Plain Old Java Object]) classes need fields, constructor, getters, equals(), hashCode(), toString().
   * Records bundle all that into a single declaration.
2. **Immutable by default**
   * Record components are final; you get a fully immutable data type out of the box.
3. **Semantic clarity**
   * Declares “this class is just data”—no hidden behavior.

**✅ When to Use Records**

* **Simple data aggregates**: DTOs, value objects, tuples, keys.
* **Immutable snapshots**: configuration settings, read‑only API responses.
* **Pattern matching & switch**: Records pair nicely with sealed classes and pattern matching (Java 17+).

**Avoid** when:

* You need mutable state.
* You need to extend another class (records implicitly extend java.lang.Record).
* You need to add significant behavior beyond accessors (though you can add methods).

**🌍 Real‑World Examples**

**1. Person Record**

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

* Auto‑generates:
  + A public constructor Person(String, int)
  + String name(), int age()
  + equals(), hashCode(), toString()

**2. Database Configuration**

public record DbConfig(String url, String user, String password) {

public DbConfig {

Objects.requireNonNull(url, "url");

Objects.requireNonNull(user, "user");

}

}

* Canonical constructor validation built‑in.

**3. Geometric Point**

public record Point(double x, double y) {

public double magnitude() {

return Math.hypot(x, y);

}

}

* You can still add custom methods.

**📋 Key Rules & Notes**

| **Rule** | **Behavior** |
| --- | --- |
| Components are final | Immutable fields |
| Implicit members | Canonical constructor, getters, equals, hashCode, toString |
| Cannot extend other classes | They implicitly extend java.lang.Record |
| Can implement interfaces | e.g., record R(int x) implements Serializable {} |
| Custom constructors & methods allowed | But must delegate to canonical constructor |

**🤔 MCQs: Test Your Records Knowledge**

1. **Which Java version fully standardized Records?**  
   A) Java 14 B) Java 15 C) Java 16 D) Java 17

**Answer:** C) Java 16

1. **What does a record declaration automatically provide?**  
   A) Mutable setters B) equals(), hashCode(), toString() C) Database mapping D) Thread safety

**Answer:** B) equals(), hashCode(), toString()

1. **Which of the following is legal?**

public record Book(String title, String author) {

public Book {

Objects.requireNonNull(title);

}

public void setTitle(String t) { … }

}

A) Valid record with setter  
B) Invalid—records can’t have setters  
C) Invalid—records can’t validate in constructor  
D) Valid—records allow setters and validation

**Answer:** B) Invalid—records can’t have setters

1. **Can a record extend another class?**  
   A) Yes  
   B) No  
   C) Only if that class is Serializable  
   D) Only in Java 17+

**Answer:** B) No

1. **Which signature matches the canonical constructor of record Point(int x, int y)?**  
   A) public Point(int x, int y)  
   B) public Point()  
   C) public Point(int y, int x)  
   D) private Point(int x, int y)  
     
   **Answer:** A) public Point(int x, int y)

# **Java Real-World Assignment: Employee Management System**

**Objective**

Create an **Employee Management System** that allows users to manage employee records, generate reports, and perform various operations using modern Java features. The system should leverage the following concepts:

* **Private methods in interfaces**
* **Collectors API & Immutable Collections**
* **var keyword**
* **Switch expressions**
* **Text blocks**
* **Records**

**Requirements**

**1. Define Employee Data Using Records**

Create an **Employee** record with the following fields:

* id (String)
* name (String)
* department (String)
* salary (double)
* joiningDate (LocalDate)

public record Employee(String id, String name, String department, double salary, LocalDate joiningDate) {}

**2. Implement an Interface with Private Methods**

Create an interface **EmployeeOperations** with:

* A default method getFormattedEmployeeInfo(Employee emp) that formats employee details using a **text block**.
* A private helper method formatDate(LocalDate date) to format the joining date.

public interface EmployeeOperations {

default String getFormattedEmployeeInfo(Employee emp) {

return """

Employee ID: %s

Name: %s

Department: %s

Salary: $%.2f

Joined On: %s

""".formatted(emp.id(), emp.name(), emp.department(), emp.salary(), formatDate(emp.joiningDate()));

}

private String formatDate(LocalDate date) {

return date.format(DateTimeFormatter.ofPattern("MMM dd, yyyy"));

}

}

**3. Use Collectors API & Immutable Collections**

Create a class **EmployeeManager** that:

* Maintains a list of employees.
* Provides methods to:
  + Filter employees by department (**using Collectors.groupingBy**).
  + Get the top 3 highest-paid employees (**using Collectors.toList with sorting**).
  + Return an **immutable list** of all employees.

import java.util.\*;

import java.util.stream.Collectors;

public class EmployeeManager implements EmployeeOperations {

private final List<Employee> employees;

public EmployeeManager(List<Employee> employees) {

this.employees = new ArrayList<>(employees);

}

public Map<String, List<Employee>> groupByDepartment() {

return employees.stream()

.collect(Collectors.groupingBy(Employee::department));

}

public List<Employee> getTop3HighestPaid() {

return employees.stream()

.sorted(Comparator.comparingDouble(Employee::salary).reversed())

.limit(3)

.collect(Collectors.toUnmodifiableList()); // Immutable list

}

public List<Employee> getAllEmployees() {

return Collections.unmodifiableList(employees); // Immutable list

}

}

**4. Use var Keyword for Local Variables**

In the **main** method, demonstrate the use of var for cleaner variable declarations.

public class Main {

public static void main(String[] args) {

var employees = List.of(

new Employee("E101", "Alice", "HR", 65000, LocalDate.of(2020, 5, 15)),

new Employee("E102", "Bob", "IT", 85000, LocalDate.of(2019, 8, 20)),

new Employee("E103", "Charlie", "Finance", 75000, LocalDate.of(2021, 3, 10)),

new Employee("E104", "Diana", "IT", 90000, LocalDate.of(2018, 11, 5))

);

var manager = new EmployeeManager(employees);

var topEmployees = manager.getTop3HighestPaid();

var hrEmployees = manager.groupByDepartment().get("HR");

System.out.println("Top 3 Employees:");

topEmployees.forEach(e -> System.out.println(manager.getFormattedEmployeeInfo(e)));

}

}

**5. Use Switch Expressions for Department-Based Bonuses**

Add a method in **EmployeeManager** that calculates a bonus based on the department using **switch expressions**:

* **IT**: 15% bonus
* **Finance**: 10% bonus
* **HR**: 5% bonus
* **Others**: 3% bonus

public double calculateBonus(Employee emp) {

return switch (emp.department()) {

case "IT" -> emp.salary() \* 0.15;

case "Finance" -> emp.salary() \* 0.10;

case "HR" -> emp.salary() \* 0.05;

default -> emp.salary() \* 0.03;

};

}

**Expected Output**

When running the Main class, the output should display:

1. **Top 3 highest-paid employees** in a formatted text block.
2. **Department-wise grouping** (optional: print HR employees).
3. **Bonus calculation** for a sample employee using switch expressions.

Top 3 Employees:

Employee ID: E104

Name: Diana

Department: IT

Salary: $90000.00

Joined On: Nov 05, 2018

Employee ID: E102

Name: Bob

Department: IT

Salary: $85000.00

Joined On: Aug 20, 2019

Employee ID: E103

Name: Charlie

Department: Finance

Salary: $75000.00

Joined On: Mar 10, 2021