

Java 16 features

Java 16 features

Java 16, released in **March 2021**, is a **short-term (non-LTS)** release as part of the six-month release cadence by Oracle. Despite being non-LTS, Java 16 introduced several important **language enhancements**, **runtime improvements**, and **new tools**.

Here is a detailed list of **all key standard features** of **Java 16**, along with explanations and code examples wherever applicable:


✓ 1. JEP 338: Vector API (Incubator)

Purpose: Introduce an API to express vector computations that compile reliably at runtime to optimal vector instructions.

```
java

import jdk.incubator.vector.*;

public class VectorExample {
    public static void main(String[] args) {
        FloatVector va = FloatVector.fromArray(FloatVector.SPECIES_256, new float[]{1, 2, 3, 4},
0);
        FloatVector vb = FloatVector.fromArray(FloatVector.SPECIES_256, new float[]{5, 6, 7, 8},
0);
        FloatVector vc = va.add(vb);
        vc.toArray(new float[4], 0);
    }
}
```

 **Incubator:** Indicates an API is not finalized. Users must explicitly enable it using `--add-modules jdk.incubator.vector`.

✓ 2. JEP 376: ZGC: Concurrent Thread-Stack Processing

Purpose: Improve performance of the **Z Garbage Collector** by processing thread stacks concurrently.

- **ZGC** now handles stack scanning **concurrently**, reducing pause times.
-

✓ 3. JEP 387: Elastic Metaspace

Purpose: Return unused HotSpot class metadata memory (Metaspace) to the OS to improve memory efficiency.

- Frees up **Metaspace memory** without JVM restart.
 - Helps in **containerized environments** and microservices.
-

✓ 4. JEP 394: Pattern Matching for instanceof (Standard)

Purpose: Simplifies `instanceof` checks with pattern binding.

```
java

Object obj = "Hello, Java 16!";
if (obj instanceof String s) {
    System.out.println(s.toUpperCase()); // No cast needed!
}
```

✓ Finalized in Java 16 (was preview in Java 14/15)

✓ 5. JEP 395: Records (Standard)

Purpose: Simplifies creation of data-carrying classes (DTOs).

```
java

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

Person p = new Person("Alice", 30);
System.out.println(p.name()); // Alice
```

✓ Finalized in Java 16 (was preview in Java 14/15)

✓ 6. JEP 396: Strong Encapsulation of JDK Internals

Purpose: Prevent reflective access to internal JDK APIs by default.

- Only `sun.misc.Unsafe` remains accessible for now.
 - You must use `--add-opens` to open modules explicitly.
-

✓ 7. JEP 397: Sealed Classes (Second Preview)

Purpose: Restricts which classes can extend or implement a class/interface.

```
java

public sealed class Shape permits Circle, Rectangle {}

final class Circle extends Shape {}
final class Rectangle extends Shape {}
```

✓ **Preview feature** (must be enabled with `--enable-preview`)

✓ 8. JEP 392: Packaging Tool (Standard)

Purpose: Provides a cross-platform tool to **package Java applications** into native installers (.exe, .pkg, .deb, etc.)

```
bash

jpackage --name MyApp --input target/ --main-jar myapp.jar --type exe
```

✓ Final version of the `jpackage` tool (was incubator in JDK 14 & 15)

✓ 9. JEP 393: Foreign Linker API (Incubator)

Purpose: Simplify calling native code (e.g., C functions) from Java.

```
java

// Simplified usage with Foreign Linker API
```

✓ Incubator module: `jdk.incubator.foreign`

✓ 10. JEP 386: Alpine Linux Port

Purpose: Adds support for running Java on **Alpine Linux** and **musl libc**.

- Useful for containerized, lightweight deployments (e.g., Docker with Alpine base images).

✓ 11. JEP 390: Warnings for Value-Based Classes

Purpose: Emit warnings when misuse of **value-based classes** like `Optional`, `LocalDate`, etc., occurs (e.g., via `synchronized`).

```
java
```

```
Optional<String> opt = Optional.of("hello");  
synchronized (opt) {} // 🚨 Will generate a warning
```

✓ 12. JEP 389: Foreign Function & Memory API (Incubator)

Purpose: Allow Java to interoperate with native code and memory safely and efficiently.

✓ Still incubator, in package: `jdk.incubator.foreign`

✓ Summary Table

JEP	Feature	Type
338	Vector API	Incubator
376	ZGC Thread-Stack Processing	Runtime
387	Elastic Metaspace	Runtime
394	Pattern Matching for instanceof	Standard
395	Records	Standard
396	Strong Encapsulation	JVM

JEP	Feature	Type
397	Sealed Classes (2nd Preview)	Preview
392	jpackage Tool	Standard
393	Foreign Linker API	Incubator
386	Alpine Linux Port	Port
390	Value-Based Class Warnings	JVM
389	Foreign Function & Memory API	Incubator

Let me know if you'd like to try some of these in code examples, or want a breakdown of only **language features**, **JVM changes**, or **tooling** separately.

you are missing Day period support, new stream methods

You're right — Java 16 also introduced **API-level enhancements** that were **not part of JEPs**, like:

✓ Additional Standard Library Enhancements in Java 16

◆ 1. New `Stream` API Methods

Java 16 added **two new methods** to the `Stream` interface:

a) `Stream.toList()`

- Collects elements into an **unmodifiable List**.

```
java
```

```
List<String> list = Stream.of("a", "b", "c").toList();  
System.out.println(list); // [a, b, c]
```

```
list.add("d"); // ❌ UnsupportedOperationException
```

📌 Internally equivalent to:

```
collect(Collectors.toUnmodifiableList())
```

b) `Stream.mapMulti(BiConsumer)`

- More efficient version of `flatMap` for avoiding intermediate streams.

java

```
List<Integer> numbers = List.of(1, 2, 3);
```

```
List<Integer> result = numbers.stream()
    .<Integer>mapMulti((num, consumer) -> {
        consumer.accept(num);
        consumer.accept(num * 10);
    }).toList();
```

```
System.out.println(result); // [1, 10, 2, 20, 3, 30]
```

✅ `mapMulti()` avoids creating temporary streams unlike `flatMap()`.

♦ 2. `java.time.format.DateTimeFormatter` - Day Period Support

New pattern letter: `B` - Day Period (like "in the morning", "in the evening")

java

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("h B");
```

```
String formatted = LocalTime.of(9, 0).format(formatter);
System.out.println(formatted); // 9 in the morning
```

```
formatted = LocalTime.of(21, 0).format(formatter);
System.out.println(formatted); // 9 in the evening
```

🕒 This provides more human-friendly **localized representations** of time of day.

Summary of Non-JEP Additions

Feature	Description
<code>Stream.toList()</code>	Returns unmodifiable list from a stream
<code>Stream.mapMulti()</code>	Efficient flat-mapping without creating intermediate streams
<code>DateTimeFormatter</code> <code>'B'</code>	Adds localized day periods like "in the morning"

Would you like a full downloadable list or code file with all these Java 16 features consolidated for quick reference?

give more details about Pattern Matching for `instanceof`

Certainly! Let's dive deep into **Pattern Matching for `instanceof`**, which was **standardized in Java 16 (JEP 394)** after being a preview in Java 14 and 15.

✅ What is Pattern Matching for `instanceof`?

📌 Before Java 16

To check the type of an object and cast it, you had to do both manually:

```
java

Object obj = "Hello Java";

if (obj instanceof String) {
    String s = (String) obj; // Explicit cast
    System.out.println(s.toUpperCase());
}
```

This was verbose, redundant, and error-prone.

✓ With Java 16: Simplified and Safer

Now, Java supports **pattern matching** with `instanceof`, allowing you to:

- Test the type
- **Declare and bind** a new variable of that type — all in one step.

```
java

Object obj = "Hello Java";

if (obj instanceof String s) {
    System.out.println(s.toUpperCase()); // No cast needed
}
```

➔ `s` is **only in scope** inside the `if` block.

✓ Key Benefits

- **No explicit casting**
 - **Safer:** Avoids `ClassCastException`
 - **Cleaner, more readable code**
 - Enables **future extension** to more powerful pattern matching (like deconstruction)
-

✓ Scope Rules for Pattern Variable

The pattern variable is:

- **Only accessible within the `if` block**
- **Not accessible in the else block or outside**

```
java

Object obj = "Example";

if (obj instanceof String s) {
    System.out.println("Inside if: " + s);
} else {
    // s is NOT accessible here
}
```

✓ Pattern Matching with Logical Operators

You can combine with other conditions:

```
java

if (obj instanceof String s && s.length() > 5) {
    System.out.println("Long string: " + s);
}
```

Here:

- `s` is defined only if both conditions are true
- Short-circuiting behavior ensures `s` is safe to use

✓ Nested Usage in Methods

You can even return values using pattern variables cleanly:

```
java

public String format(Object o) {
    if (o instanceof Integer i) return "int: " + (i + 1);
    if (o instanceof String s) return "str: " + s.toUpperCase();
    return "unknown";
}
```

⚠ Limitations in Java 16

Pattern matching in Java 16 is **limited to** `instanceof`.

Advanced features like:

- **Pattern matching in `switch` statements**
- **Record deconstruction patterns**

...were introduced **later** (in Java 17+ previews and Java 21 standard).

✓ Internal Compilation

The compiler internally rewrites:

```
java
```

```
if (obj instanceof String s) {  
    System.out.println(s);  
}
```

As:

```
java
```

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

So it's **syntactic sugar** + **type safety enforcement**.



Summary

Feature	Description
<code>instanceof</code> pattern	Combines type check + cast
Syntax	<code>if (obj instanceof Type var)</code>
Scope	Variable exists only inside the condition block
Safety	Avoids class cast exceptions
Clean	Reduces boilerplate code significantly

Let me know if you'd like examples using pattern matching in a utility method, or how this evolves in Java 17+ with `switch` **patterns** and **record deconstruction**.

give more information about stream enhancements
also

Absolutely! In **Java 16**, the `Stream` API saw important enhancements, especially with the addition of:

✓ 1. `Stream.toList()` — New Terminal Operation

◆ Description

A new terminal method that returns an **unmodifiable** list directly from a stream.

✓ Syntax

```
java  
  
List<T> toList();
```

✓ Example

```
java  
  
List<String> names = Stream.of("Alice", "Bob", "Charlie").toList();  
System.out.println(names); // [Alice, Bob, Charlie]  
  
// Attempt to modify the list  
names.add("Dave"); // ✗ Throws UnsupportedOperationException
```

🔍 Under the Hood

Equivalent to:

```
java  
  
.collect(Collectors.toUnmodifiableList());
```

But more readable and efficient. It's implemented internally with optimizations.

✓ 2. `Stream.mapMulti()` — Efficient Flat Mapping

◆ Description

An alternative to `flatMap()` but more **performant**, as it avoids creating intermediate streams.

✓ Syntax

java

```
<R> Stream<R> mapMulti(BiConsumer<? super T, ? super Consumer<R>> mapper)
```

- `mapper` : a function that accepts an element `T` and a `Consumer<R>` to emit 0 or more elements.

✓ Example 1 – Replacing flatMap

java

```
List<Integer> nums = List.of(1, 2, 3);
```

```
List<Integer> doubled = nums.stream()
    .<Integer>mapMulti((n, out) -> {
        out.accept(n);
        out.accept(n * 10);
    })
    .toList();
```

```
System.out.println(doubled); // [1, 10, 2, 20, 3, 30]
```

✓ Example 2 – Conditional Emission

java

```
List<String> words = List.of("java", "", "stream", "");
```

```
List<String> nonEmpty = words.stream()
    .<String>mapMulti((word, out) -> {
        if (!word.isEmpty()) out.accept(word);
    })
    .toList();
```

```
System.out.println(nonEmpty); // [java, stream]
```

✓ Why use `mapMulti()` over `flatMap()` ?

<code>flatMap()</code>	<code>mapMulti()</code>
Creates and flattens intermediate streams	Avoids intermediate streams
Less efficient in memory/performance	More memory and CPU efficient
Requires <code>.stream()</code> calls	Uses simple callbacks

✓ Summary Table: Java 16 Stream Enhancements

Method	Description	Return Type	Mutability
<code>toList()</code>	Collects elements into an unmodifiable list	<code>List<T></code>	Unmodifiable
<code>mapMulti()</code>	Efficient mapping of 0+ elements per input	<code>Stream<R></code>	Depends on downstream collector

✓ Practical Use Cases

Use Case	Solution
Filter and transform	<code>filter()</code> + <code>map()</code> or <code>mapMulti()</code>
Expand values (e.g. <code>x -> x, x*2</code>)	<code>mapMulti()</code>
Conditional flattening	<code>mapMulti()</code> with <code>if</code> inside

Would you like a side-by-side comparison code of `flatMap()` vs `mapMulti()` or a PDF/Markdown cheat sheet with all Java 16 features including Stream and language enhancements?

Certainly! Let's clearly compare **before Java 16 (without `.toList()`)** and **with Java 16+ (using `.toList()`)**, both in terms of:

- Code syntax
 - Behavior
 - Mutability
 - Readability
-

✓ Use Case: Collect elements from a stream into a `List`

◆ Before Java 16 (e.g., Java 8 to 15)

You had to use:

```
java

import java.util.stream.Collectors;

List<String> list = Stream.of("a", "b", "c")
    .collect(Collectors.toList()); // modifiable list
```

Or if you wanted an **unmodifiable list**, it got even longer:

```
java


List<String> list = Stream.of("a", "b", "c")
    .collect(Collectors.toUnmodifiableList()); // Java 10+
```

- Verbose and not very readable
 - Easy to forget that `.toList()` from `Collectors` is **modifiable**
-

◆ Problems Before `.toList()`

1. **Unclear mutability:** `Collectors.toList()` returns a mutable list.
2. **Verbosity:** `Collectors.toUnmodifiableList()` is long to type.
3. **Risk of unintended modifications.**

```
java
```

```
List<String> modifiableList = Stream.of("x", "y").collect(Collectors.toList());  
modifiableList.add("z"); //  allowed
```

```
List<String> unmodifiableList = Stream.of("x", "y").collect(Collectors.toUnmodifiableList());  
unmodifiableList.add("z"); //  throws UnsupportedOperationException
```

With Java 16+: `Stream.toList()`


Simplified:

```
java
```

```
List<String> list = Stream.of("a", "b", "c").toList();
```

- Returns an **unmodifiable List**
- No need for `Collectors`
- Safer and cleaner syntax

```
java
```

```
list.add("d"); //  throws UnsupportedOperationException
```

Comparison Table

Feature	<code>collect(Collectors.toList())</code>	<code>collect(Collectors.toUnmodifiableList())</code>	<code>Stream.toList()</code> (Java 16+)
Returns	Modifiable List	Unmodifiable List	Unmodifiable List
Import Needed	<code>java.util.stream.Collectors</code>	Yes	No
Syntax	Verbose	Very Verbose	Concise
Thread-Safety	Not thread-safe	Not thread-safe	Not thread-safe
Java Version	8+	10+	16+

✓ Recommended Usage

- Use `.toList()` by default in Java 16+ when you want:
 - Simpler code
 - Immutable result
 - Less risk of accidental modification
-

🔧 Example: Before vs After

🔴 Before Java 16

java

```
List<String> names = Stream.of("Tom", "Jerry", "Spike")
    .collect(Collectors.toUnmodifiableList());
```

🟢 Java 16+

java

```
List<String> names = Stream.of("Tom", "Jerry", "Spike").toList();
```

Same result, cleaner and safer!

Let me know if you'd like examples of combining `.toList()` with `filter()`, `map()`, or `mapMulti()` !

Record explain all functionalities, like add Record add in inner class

Absolutely! Let's deep dive into **Records in Java**, introduced as a **preview in Java 14**, finalized in **Java 16 (JEP 395)**.

✓ What is a Record in Java?

A **record** is a special **type of class** in Java that is a **concise way to model immutable data**. It:

- Is implicitly `final`
- Automatically provides:
 - `private final` fields
 - Constructor
 - `equals()`, `hashCode()`, `toString()`
 - Getters (called *accessor methods*)

✓ Basic Syntax

java

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




This is **equivalent to**:

java

```
public final class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) { ... }  
  
    public String name() { return name; }  
    public int age() { return age; }  
  
    public boolean equals(Object o) { ... }  
    public int hashCode() { ... }  
    public String toString() { ... }  
}
```

✓ Key Features of Records

Feature	Description
Immutable fields	All fields are <code>private final</code>

Feature	Description
Canonical constructor	Auto-generated with all fields
Auto-generated methods	<code>equals()</code> , <code>hashCode()</code> , <code>toString()</code>
Compact constructor	You can define your own validation
Accessors	Methods named after fields (<code>.name()</code>) instead of <code>.getName()</code>)
Can implement interfaces	Yes 
Cannot extend classes	No  inheritance (records are <code>final</code>)
Can be nested (inner class)	Yes 

Example: Using a Record

java

```
public record User(String username, String email) {}
```

```
User u = new User("admin", "admin@email.com");
```

```
System.out.println(u.username()); // admin
```

```
System.out.println(u); // User[username=admin, email=admin@email.com]
```

Custom Constructor in Record

Compact Constructor

java

```
public record Product(String name, double price) {
    public Product {
        if (price < 0) throw new IllegalArgumentException("Price must be positive");
    }
}
```

Note: You **don't need to specify parameters again** — it's called a **compact canonical constructor**.

Explicit Canonical Constructor

java

```
public record Product(String name, double price) {
    public Product(String name, double price) {
        if (price < 0) throw new IllegalArgumentException("Invalid price");
        this.name = name;
        this.price = price;
    }
}
```

Non-canonical Constructor (Overloaded)

java

```
public record Rectangle(int width, int height) {
    public Rectangle(int size) {
        this(size, size); // Overloaded constructor
    }
}
```

Record with Methods

java

```
public record Circle(double radius) {
    public double area() {
```

```
    return Math.PI * radius * radius;
}
}
```

✓ Records Can Implement Interfaces

```
java

interface Identifiable {
    String id();
}

public record Employee(String id, String name) implements Identifiable {}
```

✓ Records as Inner Classes

Yes, records can be defined inside classes, including:

1. Static Inner Record

```
java

public class Outer {
    public static record Point(int x, int y) {}
}
```

Usage:


```
java


Outer.Point p = new Outer.Point(10, 20);
```

2. Non-static Inner Record ❌ Not Allowed

- Records cannot be non-static inner classes.
- Java will throw a compiler error if you try:

```
java
```

```
public class Outer {  
    public record Inner(int value) {} //  implicitly static  
}
```

 Inner records are always **implicitly static** — same as static nested classes.

Record Inside a Method (Local Record)

Java 16+ also supports **local records**:

```
java  
  
public void process() {  
    record Entry(String key, String value) {}  
  
    Entry e = new Entry("foo", "bar");  
    System.out.println(e);  
}
```

Useful for short-lived data holders in methods.

Limitations of Records

Limitation	Reason
No setters	Records are immutable
Cannot extend other classes	Record implicitly extends <code>java.lang.Record</code>
Fields are always final	Enforces immutability
Cannot define instance fields (other than components)	Keeps data compact and consistent

When to Use Records

- Data Transfer Objects (DTOs)
- Value objects (like coordinates, tuples)
- Response wrappers (e.g., `ApiResponse(status, data)`)

✓ Real-World Example

java

```
public record ApiResponse<T>(String status, T data) {}
```

```
ApiResponse<String> res = new ApiResponse<>("success", "Record works!");  
System.out.println(res.status()); // success  
System.out.println(res.data()); // Record works!
```

← END Summary

Feature	Supported in Record?
Constructor customization	✓ (canonical, compact, overloaded)
Method definitions	✓
Implement interfaces	✓
Nesting inside a class	✓ (static only)
Inheritance from class	✗
Mutability	✗ (fully immutable)

Would you like a downloadable PDF/Markdown guide with full examples, or a comparison between **records vs classes vs Lombok @Data**?

some default methods and proxies also there in java 16 right?

Yes, you're right again! Java 16 introduced **some lesser-known but valuable enhancements** in the core libraries, including:

✓ 1. Enhancements to `java.lang.reflect.Proxy`

📌 Proxy class enhancement to support default interface methods

Java 16 fixed a long-standing issue:

Dynamic proxies can now correctly invoke default methods on interfaces.

🔍 What was the issue before?

Prior to Java 16, if you used `java.lang.reflect.Proxy` to implement interfaces with **default methods**, those default methods were not correctly invoked unless you handled them explicitly via `MethodHandles`.

✓ Now in Java 16:

The `InvocationHandler` passed to a `Proxy` can simply delegate, and default methods will work **automatically**, without special effort.

🔧 Example

```
java

import java.lang.reflect.*;

interface Greeting {
    default String hello() {
        return "Hello from default!";
    }
}
```

```

public class ProxyTest {
    public static void main(String[] args) {
        Greeting proxy = (Greeting) Proxy.newProxyInstance(
            Greeting.class.getClassLoader(),
            new Class<?>[]{Greeting.class},
            (p, method, args1) -> {
                // Just delegate to the default method if present
                if (method.isDefault()) {
                    // Java 16 handles it internally — NO special handling required
                    return InvocationHandler.invokeDefault(proxy, method, args1);
                }
                return "Handled: " + method.getName();
            }
        );

        System.out.println(proxy.hello()); // ✅ Output: Hello from default!
    }
}

```

⚠️ `InvocationHandler.invokeDefault()` is **not public API**, this is internal — in Java 16, you don't need to manually call it anymore.

✅ 2. New Default Methods in Core Interfaces

Java 16 added **new default methods** to key interfaces to improve stream and collection handling:

♦ `java.util.Map` — `orElseThrow()`

java

```
Map<String, String> map = Map.of("a", "apple");
```

```
String value = map.getDefault("b", null);
```

```
// Still null
```

```
String requiredValue = Optional.ofNullable(map.get("b"))
    .orElseThrow(() -> new NoSuchElementException("Key not found"));
```


While not a new method **on Map** itself, the pattern is supported more cleanly due to consistent use of `Optional`.

- ◆ `InputStream.transferTo(OutputStream)` – added in Java 9, still very useful and often missed

```
java

try (InputStream in = new FileInputStream("file.txt");
    OutputStream out = new FileOutputStream("copy.txt")) {
    in.transferTo(out); // Efficient, optimized file copy
}
```

- ◆ `Collection.toArray(IntFunction<T[]>)` — Added earlier but widely used in Java 16+

```
java

List<String> names = List.of("Tom", "Jerry");
String[] array = names.toArray(String[]::new);
```

- ◆ `Optional.isEmpty()` — Java 11+ feature but heavily used in Java 16 APIs

```
java

Optional<String> maybeName = Optional.empty();

if (maybeName.isEmpty()) {
    System.out.println("No name found");
}
```

✓ Summary of Small But Important Enhancements

Area	Enhancement	Java 16 Behavior
Proxy	Supports default methods	✓ Transparent support
Stream	<code>toList()</code> , <code>mapMulti()</code>	✓ Efficient & safe collection
Map + Optional	Better idioms with <code>orElseThrow()</code>	✓ Clean and functional
Collections	Improved usage with method references	✓ More readable
Reflection	Default method delegation	✓ Automatic in proxies

Would you like a mini project demonstrating dynamic proxies using interface default methods and custom logging?

create interface and default methods and explain proxies

Great! Let's **walk through step by step** how **Java dynamic proxies** work with **default methods in interfaces**, introduced in **Java 16** and enhanced to work more intuitively.

✓ Step 1: Understand the Basics

What is a Dynamic Proxy?

A **proxy** in Java is an object that **implements one or more interfaces** and **delegates method calls** to an `InvocationHandler`.

Use cases:

- Logging
- Security checks

- Caching
 - Lazy loading
-

✓ Step 2: Create an Interface with Default Methods

```
java

public interface Service {
    void doWork();

    default String getStatus() {
        return "Service is running (from default method)";
    }
}
```

- `doWork()` is an **abstract method**
 - `getStatus()` is a **default method**
-

✓ Step 3: Create a Proxy Using `Proxy.newProxyInstance`

```
java

import java.lang.reflect.*;

public class ProxyDemo {
    public static void main(String[] args) {
        Service proxy = (Service) Proxy.newProxyInstance(
            Service.class.getClassLoader(),
            new Class<?>[]{Service.class},
            new MyInvocationHandler()
        );

        proxy.doWork();           // handled by InvocationHandler
        System.out.println(proxy.getStatus()); // handled by default method
    }
}
```

✓ Step 4: Implement the `InvocationHandler`

java

```
class MyInvocationHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // Check if it's a default method
        if (method.isDefault()) {
            // As of Java 16, this works automatically. But you can handle it explicitly too:

            // Lookup the declaring class (interface)
            Class<?> declaringClass = method.getDeclaringClass();

            // Use MethodHandles to invoke default method
            return MethodHandles.lookup()
                .findSpecial(declaringClass, method.getName(),
                    MethodType.methodType(method.getReturnType(),
                        method.getParameterTypes()),
                    declaringClass)
                .bindTo(proxy)
                .invokeWithArguments(args == null ? new Object[0] : args);
        }

        // Handle normal method
        System.out.println("Proxy intercepted method: " + method.getName());
        return null;
    }
}
```

✓ **Note:** In Java 16+, default methods are correctly routed even without `MethodHandles`, but you can still use this approach manually.

✓ Output

sql

Proxy intercepted method: doWork
Service is running (from default method)

✓ How This Works

Part	Role
<code>Service</code> interface	Defines both abstract & default methods
<code>Proxy.newProxyInstance()</code>	Creates a proxy object at runtime
<code>InvocationHandler</code>	Handles method calls (like interceptors)
<code>method.isDefault()</code>	Used to detect and invoke interface default methods
<code>MethodHandles</code>	Used to reflectively call the default implementation

✓ Limitations


- Works **only with interfaces** (not concrete classes).
- Proxy can only **implement interfaces**, not extend abstract classes.
- `InvocationHandler` has no way to call `super.method()` for non-default methods (unless delegated manually).

✓ Summary

Topic	Java 16 Behavior
Proxy of default methods	✓ Supported natively
Manual <code>MethodHandles</code> for default methods	✓ Optional (mostly for custom control)
Dynamic Proxy	✓ Allows custom logic per method

Topic**Java 16 Behavior**

Interface required

 Must be an interface

Would you like to see how to add **logging**, **authorization**, or **caching** using dynamic proxies with default method support?