Features introduced in Java 8

- **Lambda Expressions:** Enable functional programming by writing functions inline.
- **Stream API:** Process collections in a functional style.
- **Functional Interfaces:** Interfaces with a single abstract method.
- **Optional**: Avoid NullPointerException.
- **Default Methods**: Add default implementations in interfaces.
- **Date and Time API**: Improved handling of dates and times.
- **Method References**: Simplified syntax for calling methods.

Lambda Expression:

Lambda expression is an Anonymous function.
1.Not having any name, Not having any return type and Not having any access modifier.

Syntax:

```
(parameters) -> { body }
```

Before Java 8 (using anonymous inner class):

An anonymous inner class (a class without a name) is used to implement superclass or interface methods, mainly for one-time use. However, it's verbose and hard to read. Java 8 introduced lambda expressions to simplify this by offering a cleaner and shorter way to implement functional interfaces, improving readability and enabling functional programming.

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Hello World");
    }
};
```

With Java 8 Lambda:

```
Runnable r = () -> System.out.println("Hello World");
```

Functional Interface + Lambda

A **Functional Interface** has only one abstract method. Lambda expressions are used to implement them.

```
Runnable runTask = () -> System.out.println("Running in a thread");

Thread thread = new Thread(runTask);
thread.start();
```

Stream API:

Stream API allows operations on collections and Arrays. It helps write concise, readable, and efficient code by chaining operations like filtering, mapping, and collecting, and it supports both sequential and parallel execution.

Why Stream API introduced?

Before Java 8:

- We used **loops** (like for, for-each) to process collections.
- These loops made code **imperative**, **verbose**, and less readable.

With Java 8 Streams:

- We **focus on "what" to do**, not "how" to do.
- Code becomes **cleaner**, **shorter**, and more **maintainable**

Type of Stream:

- **Intermediate Operations(return a Stream):**
  - filter(): Filter elements based on a condition.
  - map(): Transform elements.
  - flateMap(): Flattens a stream of streams into a single stream.

```
List<List<String>> listOfLists = Arrays.asList(
    Arrays.asList("A", "B"),
    Arrays.asList("C", "D")
);
List<String> flatList = listOfLists.stream()
                                   .flatMap(list->ist.stream())
                                   .collect(Collectors.toList());
```

  - sorted(): Sort elements.

- **Terminal Operations (consume the Stream):**
  - collect(): Converts stream to List, Set, etc.
  - forEach(): Iterates over each element.
  - reduce(): Combines elements into a single value.
  - anyMatch()/allMatch()/noneMatch() : Checks Condition.

ParallelStream(): It Allows processing of data concurrently using multiple threads. ParallelStream runs with the help of Fork Join Pool which divides the data into chunks and processes them independently and then combines into the result.

```
List<String> names = Arrays.asList("Sachin", "Rohit", "Virat", "Dhoni", "Raina");
    names.parallelStream().forEach(name -> {
            System.out.println(Thread.currentThread().getName() + " : " + name);
        });
```

What is the purpose of Collectors?

Collectors is a utility for reducing streams.

Common collectors:

- ➔ ○ **toList(),toSet():** Convert to a list or set *.collect(Collectors.toList()).*
- ➔ ○ **joining():** Concatenate strings.
- ➔ ○ **groupingBy():** Group elements by a key *Collectors.groupingBy().*
- ➔ ○ **partitioningBy():** Partition stream into two groups (true/false).

Functional Interfaces and Interface Evolution

Java 8 introduced functional interfaces to support lambda expressions and functional programming. These interfaces contain only one abstract method and can have multiple default or static methods.

Core Functional Interfaces

Function<T, R>

- Takes one input (T) and returns one output (R).
- Used for mapping or transforming data.

Example:

```
Function<String, Integer> lengthFunc = str -> str.length();
```

Consumer<T>

- Takes one input and returns nothing.
- Used to perform operations like printing or saving data.

Example:

```
Consumer<String> printer = name -> System.out.println(name);
```

Supplier<T>

- Takes no input but returns a value.
- Used for supplying/generating values.

Example:

```
Supplier<Double> randomSupplier = () -> Math.random();
```

Predicate<T>

- Takes one input and returns boolean (true/false).
- Used for filtering or validating data.

Example:

```
Predicate<Integer> isEven = num -> num % 2 == 0;
```

Additional Functional Interfaces

- **BiFunction<T, U, R>:** Takes two inputs and returns a result.
- **BiConsumer<T, U>:** Takes two inputs and returns nothing.
- **BiPredicate<T, U>:** Takes two inputs and returns a boolean.
- **UnaryOperator<T>:** A Function where input and output are the same type.
- **BinaryOperator<T>:** A BiFunction where input and output are the same type.

Evolution of Java Interfaces (Before and After Java 8)

Before Java 8

• Interfaces were 100% abstract and could only contain abstract methods and constants.
• Adding a new method to an interface broke all implementing classes.

Example:

```
interface Vehicle { void start(); }
```

Java 8 Enhancements

To allow interface evolution without breaking existing code, Java 8 introduced:

Default Methods

• Provide default implementation inside the interface without breaking the existing code or functionality.
• If a class implements the default method then the class can change the Implementation of the default method if needed.
Analogy: It's like adding a new feature to a smartphone via software update, without forcing every phone to be rebuilt

Example:

```
default void stop() { System.out.println("Stopped"); }
```

Static Methods

• Utility methods that belong to the interface itself (not instance).
• **Why:** Static methods in interfaces were introduced in Java 8 to allow us to write utility/helper methods directly inside the interface. This keeps related code in one place, making the interface more self-contained and easier to use. It avoids the need for separate utility classes and improves code organization.

Example:

```
static int square(int x) { return x * x; }
```

Why These Changes?
• Enable backward-compatible interface evolution.
• Avoid breaking existing code when APIs are updated.
• Make interfaces more powerful with shared utility methods and behavior.

Optional Class:

Optional<T> class introduced in Java 8 to handle null values safely.
OR To avoid **NullPointerExceptions**.
It either contains a non-null value or is empty.

How do you use it?

We use Optional.of() for non-null values, Optional.ofNullable() for possibly null values, and Optional.empty() when there's no value. Then we can check if a value is present using isPresent() or ifPresent() and safely access it using orElse() or get().

```
public class HandleNullCheck {

    // Returns a non-null string
    public static String getId(int id) {
        String s = "sachin";
        return s;
    }

    // Might return null
    public static String getName(int id) {
        String name = null; // simulate null value
        return name;
    }

    public static void main(String[] args) {
        String name = getName(1); // This will return null

        // ❌ This will throw NullPointerException
        System.out.println(name.toUpperCase());

        System.out.println("***********************************");

        String id = getId(10); // Returns non-null
        System.out.println(id.toUpperCase()); // ✅ Works fine
    }
}
```

✅ Method Reference

Method Reference is a **shorter way to write lambda expressions** to call a method.
It uses the :: operator to refer to a method **by its name.**

Types of Method References

1. **Static Method Reference**
   👉 Syntax: ClassName::staticMethod
   👉 Used when you want to refer to a **static method.**
   📌 *Example:* Math::sqrt

```
public static void greet() {
System.out.println("Hello (Static)!");
    }
 public static void main(String[] args) {
        Runnable r = Demo::greet;
        r.run(); // Output: Hello (Static)!
    }
```

2. **Instance Method Reference of a Particular Object**
   👉 Syntax: object::instanceMethod
   👉 Used when you already have an object and want to call its method.
   📌 *Example:* System.out::println

3. **Instance Method Reference of an Arbitrary Object of a Class**
   👉 Syntax: ClassName::instanceMethod
   👉 Used when method will be called on **each object** in a collection (like in forEach).
   📌 *Example:* String::toLowerCase

```
List<String> list = Arrays.asList("sachin", "rohit", "virat");
list.stream().map(String::toUpperCase)

.forEach(System.out::println); // Output: SACHIN, ROHIT, VIRAT
```

4. **Constructor Reference**
   👉 Syntax: ClassName::new
   👉 Used when you want to refer to a **constructor** to create objects.
   📌 *Example:* ArrayList::new

✅ Date and Time API

🔹 Why Java 8 introduced a new Date and Time API?

- The **old API (java.util.Date, Calendar)** was:
  - **Mutable** – not thread-safe.
  - **Poorly designed** – confusing method names and behaviors.
  - **No clear time zone or formatting support.**

🔄 Java 8 introduced the new **java.time** package to fix all these problems with a **clean, immutable, thread-safe design.**

✅ Commonly Used Classes (in java.time package)

1. LocalDate

- Represents a date (yyyy-MM-dd) without time or timezone.

```
import java.time.LocalDate;
public class Demo {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalDate birthday = LocalDate.of(2000,05,29);
        System.out.println("Today: " + today);
        System.out.println("Birthday: " + birthday);
    }
}
```

---

2. LocalTime

- Represents only time (hh:mm:ss).

```
import java.time.LocalTime;
public class Demo {
    public static void main(String[] args) {
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);
    }
}
```

---

3. LocalDateTime

- Combines date + time but still no timezone.

```
import java.time.LocalDateTime;
public class Demo {
        public static void main(String[] args) {
          LocalDateTime now = LocalDateTime.now();
          System.out.println("Now: " + now);
        }
      }
```

---

4. ZonedDateTime

- Date and time with a timezone.

```
import java.time.ZonedDateTime;
public class Demo {
    public static void main(String[] args) {
        ZonedDateTime now = ZonedDateTime.now();
        System.out.println("Zoned DateTime: " + now);
    }
}
```

---

5. Period and Duration

- Period is used for date-based amounts (e.g., 2 years, 3 months).
- Duration is used for time-based amounts (e.g., 5 hours, 20 seconds).

```
import java.time.LocalDate;
import java.time.Period;
public class Demo {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalDate futureDate = LocalDate.of(2025, 12, 25);
        Period period = Period.between(today, futureDate);
        System.out.println("Period: " + period.getYears() + " years, " + period.getMonths() +
" months");
    }
}
```

Important packages for the new Date and Time API?

- java.time
    - dates
    - times
    - Instants
    - durations
    - time-zones
    - periods
- Java.time.format
- Java.time.temporal
- java.time.zone

✅ Top Java 11 Features (Interview-Focused)

🔹 1. `var` in Lambda Parameters

What? You can now use `var` to declare lambda parameters (introduced in Java 10, enhanced in 11).
Why? Useful for annotations or better readability.

✅ **Example:**

```
list.forEach((var name) -> System.out.println(name.toUpperCase()));
```

🔹 2. New String Methods

Java 11 added several useful methods in the **`String`** class.

✅ **Important ones:**

- **`isBlank()`** → Returns true if the string is empty or contains only white spaces.
- **`lines()`** → Converts string into a stream of lines.
- **`strip()`** → Removes leading/trailing white spaces (Unicode-aware).
- **`stripLeading()`, `stripTrailing()`** → Removes whitespaces from start/end.
- **`repeat(int count)`** → Repeats the string n times.

✅ **Examples:**

```
System.out.println(" ".isBlank()); // true
System.out.println(" Java\n11\nRocks ".lines().count()); // 3 lines
System.out.println("  hello  ".strip()); // "hello"
System.out.println("Hi".repeat(3)); // HiHiHi
```

🔹 3. `Optional.isEmpty()`

What? Checks if an **`Optional`** is empty (instead of using `!isPresent()`).

✅ **Example:**

```
Optional<String> opt = Optional.empty();
if (opt.isEmpty()) {
    System.out.println("No value present");
}
```

🔹 4. New `HttpClient` (Standardized)

Java 11 introduces a new, more powerful HTTP client in `java.net.http`.

✅ **Why?** Replaces legacy `HttpURLConnection` for making HTTP requests.

✅ **Example:**

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://api.example.com"))
        .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

🔹 5. Running Java Files without Compilation

Java 11 allows you to run `.java` files directly from the command line — like scripting.

✅ **Example:**

```
java Hello.java
(No need to compile with javac first!)
```

🔹 6. Collection `toArray(IntFunction<T[]>)` Improvement

Why? Safer, cleaner way to convert to arrays.

✅ **Example:**

```
List<String> list = List.of("A", "B", "C");
String[] arr = list.toArray(String[]::new); // instead of new
String[list.size()]
```

🔹 7. Removed and Deprecated Features

Some APIs were removed or deprecated in Java 11 (not asked often, but worth knowing):

- **Removed:** Java EE modules like `javax.xml.bind`, `javax.activation`

- **Deprecated:** Nashorn JavaScript engine

END Summary Table (Quick View)

Feature	Why It Matters	Example
`var` in lambda	Better readability/annotations	`(var x) -> x*x`
`String` methods	Useful string handling improvements	`" ".isBlank()`
`Optional.isEmpty()`	Cleaner null handling	`opt.isEmpty()`
`HttpClient`	REST APIs in Java	`HttpClient.newHttpClient()`
Run `.java` without compile	Saves time for small/test code	`java Hello.java`
`toArray()` improvement	Type-safe conversion to array	`list.toArray(String[]::new)`

✅ Java 17 Features

🔹 1. Sealed Classes

What? Sealed classes restrict which other classes can extend or implement them.

Why? To provide **better security and maintainability** by **restricting inheritance**.

✅ **Example:**

```
public sealed class Animal permits Dog, Cat {}

final class Dog extends Animal {}
final class Cat extends Animal {}
```

🧠 Think of it like: "Only selected children allowed to extend the parent."

🔹 2. Pattern Matching for instanceof

What? No need to cast after using instanceof.

Why? Reduces boilerplate and improves readability.

✅ **Before Java 17:**

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

✅ **Now (Java 17):**

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

🔹 3. Switch Expressions (Enhanced)

What? switch can return values and use multiple labels(Case) & support ->.

Why? Makes switch **shorter, safer, and expressive.**

✅ **Example:**

```
String result = switch (day) {

    case "MON", "TUE" -> "Weekday";
    case "SAT", "SUN" -> "Weekend";
    default -> "Invalid";
};
```

🔹 4. Text Blocks (Since Java 15, widely used in Java 17)

What? Multi-line strings using """.

Why? Clean, readable format for HTML, JSON, SQL, etc.

✅ **Example:**

```
String html = """
    <html>
        <body>Hello</body>
    </html>
    """;
```

🔹 5. Records (Standardized in Java 16, used in Java 17)

What? A compact way to create immutable data classes.

Why? Avoids boilerplate for getters, constructors, toString, etc.

✅ **Example:**

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

Person p = new Person("Sachin", 25);
System.out.println(p.name()); // Sachin
```

🔹 6. New Methods in Classes (Bonus for Coding)

✅ **String .stripIndent()**
Removes common indentation in multi-line strings.

✅ **Optional .ifPresentOrElse()**

```
opt.ifPresentOrElse(
    val -> System.out.println(val),
    () -> System.out.println("No value")
);
```

✅ Java 21 Features

1. Virtual Threads (Final) – JEP 444

- Allows creating thousands/millions of lightweight threads.
- Great for high-concurrency tasks like web servers.

```
Thread.startVirtualThread(() -> {
    System.out.println("Running in virtual thread: " +
Thread.currentThread());
});
```

---

2. Record Patterns (Final) – JEP 440

- Simplifies pattern matching with records.

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

Object obj = new Person("Sachin", 25);
if (obj instanceof Person(String name, int age)) {
    System.out.println("Name: " + name + ", Age: " + age);
}
```

---

3. Pattern Matching for `switch` (Final) – JEP 441

- Allows using `switch` with types and patterns, making it more powerful.

```
static String format(Object obj) {
    return switch (obj) {
        case Integer i -> "Integer: " + i;
        case String s  -> "String: " + s;
        default        -> "Unknown";
    };
}
```

---

4. Sequenced Collections – JEP 431

- New interfaces: `SequencedCollection`, `SequencedSet`, `SequencedMap`.
- Maintains a **defined encounter order** for consistent element access.

Example:

```
SequencedSet<String> set = new LinkedHashSet<>();
set.add("one");
set.add("two");
System.out.println(set.getFirst()); // one
System.out.println(set.getLast());  // two
```

5. String Interpolation

To **make string concatenation cleaner**
To **avoid messy + operators** or String.format() calls

Write cleaner and safe string interpolation.

```
String name = "Sachin";
String result = STR."Hello \{name}";
—---------------------------------
int a = 5, b = 10;
String result = STR."Sum of \{a} + \{b} = \{a + b}";
System.out.println(result);  // Sum of 5 + 10 = 15
```