Triumphant Institute of Management Education Pvt.Ltd.

Advance Educational Activities Pvt. Ltd.

Unit 6: Java 8 Features

6.1.1 Java 8 Features

Java 8 introduced powerful features that made Java more **concise**, **functional**, and **efficient**, especially for working with data and behavior.

1. Lambda Expressions

- Enables writing **anonymous functions** in a concise way.
- Makes code **shorter and cleaner**, especially with collections and threads.

2. Functional Interfaces

- Interfaces with a **single abstract method**, used with lambdas.
- Examples: Runnable, Callable, Comparator.

3. Streams API

- Processes collections using **functional-style operations**.
- Supports methods like filter(), map(), collect(), reduce().

4. Default and Static Methods in Interfaces

- Interfaces can now have **method bodies** (default or static).
- Enables backward compatibility with interface enhancements.

5. Method References

- A **shorthand** for calling methods via lambdas.
- Example: System.out::println.

6. Optional Class

- Helps handle **null values** safely.
- Avoids NullPointerException using isPresent(), orElse(), etc.

7. New Date and Time API

- java.time package provides **modern, immutable**, and **thread-safe** classes like LocalDate, LocalTime, and Period.

8. Collectors

- Used with Streams to **gather results**.
- Example: collect(Collectors.toList()).

6.2.1 Lambda Expressions in Java 8

1. What is a Lambda Expression?

A **lambda expression** is a concise way to represent an **anonymous function** (i.e., a function without a name) that can be passed as an argument or used to implement a **functional interface**.

It provides a **clear and simple syntax** for writing inline behavior.

2. Syntax

(parameters) -> { body }

Variants:

TYPE	EXAMPLE
NO PARAMETER	() -> System.out.println("Hello")
ONE PARAMETER	x -> x * x
MULTIPLE PARAMETERS	(a, b) -> a + b
WITH DATA TYPE (OPTIONAL)	(int a, int b) -> a * b
WITH BLOCK AND RETURN STATEMENT	(a, b) -> { return a + b; }

3. When to Use Lambda?

- To implement **functional interfaces** (interfaces with a single abstract method).
- Common with APIs like:
  - Runnable
  - Comparator
  - ActionListener
  - Stream operations

4. Example: Using Lambda with Runnable

Without Lambda:

```
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Thread running");
    }
};
new Thread(r1).start();
```

With Lambda:

```
Runnable r2 = () -> System.out.println("Thread running");
new Thread(r2).start();
```

5. Example: Custom Functional Interface

```
@FunctionalInterface
interface Calculator {
    int operate(int a, int b);
}

public class LambdaDemo {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        System.out.println(add.operate(5, 3));  // Output: 8
    }
}
```

6. Lambda with Collections (Streams)

```
List<String> list = Arrays.asList("Java", "Python", "C++");

list.forEach(language -> System.out.println(language));
```

Or use method reference:

```
list.forEach(System.out::println);
```

7. Benefits of Lambda Expressions

FEATURE	BENEFIT
CONCISE	Less boilerplate code
READABLE	Clear and focused on business logic
REUSABLE	Easily pass behavior as parameters
EFFICIENT	Encourages functional programming

6.2.2 Functional Interface in Java 8

1. What is a Functional Interface?

A **Functional Interface** is an interface that contains **exactly one abstract method**.

- It can have **default** or **static methods** (with implementation), but only **one abstract method**.
- Functional interfaces can be used as the **target types** for **lambda expressions** or **method references**.

2. @FunctionalInterface Annotation

This annotation is optional but recommended.
It helps the compiler **enforce the rule** that the interface should only have **one abstract method**.

```
@FunctionalInterface
interface MyInterface {
    void show();
}
```

3. Example: Custom Functional Interface with Lambda

```
@FunctionalInterface
interface Greetable {
    void greet(String name);
}

public class Test {
    public static void main(String[] args) {
        Greetable g = name -> System.out.println("Hello, " + name);
        g.greet("Alice");
    }
}
```

4. Built-in Functional Interfaces (from java.util.function)

INTERFACE	METHOD	DESCRIPTION
PREDICATE<T>	boolean test(T)	Tests a condition and returns boolean
FUNCTION<T,R>	R apply(T)	Converts input of type T to R
CONSUMER<T>	void accept(T)	Performs action on an object
SUPPLIER<T>	T get()	Supplies a value of type T

5. Examples of Built-in Functional Interfaces

Predicate

```
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isEven.test(4));  // true
```

Function

```
Function<String, Integer> strLength = s -> s.length();
System.out.println(strLength.apply("Java"));  // 4
```

Consumer

```
Consumer<String> display = s -> System.out.println(s);
display.accept("Hello World");
```

Supplier

```
Supplier<Double> random = () -> Math.random();
System.out.println(random.get());
```

6. Functional Interface with Thread

```
Runnable r = () -> System.out.println("Running thread using lambda");
new Thread(r).start();
```

7. Why Use Functional Interfaces?

- Enables **functional programming** in Java.
- Required to use **Lambda Expressions**.
- Promotes **cleaner and more concise code**.
- Encourages **reusability of behavior**.

6.2.3 Java 8 Built-in Functional Interfaces (Deep Dive)

1. Predicate<T>

Purpose:

Represents a **boolean-valued function** of one argument.

Functional Method:

```
boolean test(T t);
```

Common Use:

Used for **filtering** and **conditional logic**.

Example:

```
Predicate<String> startsWithA = s -> s.startsWith("A");
System.out.println(startsWithA.test("Apple"));  // true
Chaining with and(), or(), negate():
Predicate<String> lengthCheck = s -> s.length() > 3;
Predicate<String> combined = startsWithA.and(lengthCheck);
System.out.println(combined.test("Ace"));  // false
```

2. Function<T, R>

Purpose:

Takes a value of type T and returns a value of type R.

Functional Method:

```
R apply(T t);
```

Common Use:

Used for **data transformation**.

Example:

```
Function<String, Integer> strToLength = s -> s.length();
System.out.println(strToLength.apply("Java"));  // 4
```

Chaining:

- andThen(): executes after current
- compose(): executes before current

```
Function<String, String> addPrefix = s -> "Hello " + s;
Function<String, String> toUpper = s -> s.toUpperCase();

System.out.println(addPrefix.andThen(toUpper).apply("john")); // HELLO JOHN
```

3. Consumer<T>

Purpose:

Accepts a value of type T and returns nothing (void).

Functional Method:

```
void accept(T t);
```

Common Use:

Used for **printing, logging, or saving** without returning a result.

Example:

```
Consumer<String> printUpper = s -> System.out.println(s.toUpperCase());
printUpper.accept("hello");  // HELLO
Chaining with andThen():
Consumer<String> printLength = s -> System.out.println(s.length());
printUpper.andThen(printLength).accept("Java");
```

4. Supplier<T>

Purpose:

Takes **no input** but **returns** a result of type T.

Functional Method:

```
T get();
```

Common Use:

Used for **generating values** like random numbers, timestamps, etc.

Example:

```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

5. BiFunction<T, U, R>

Purpose:

Takes **two inputs** of types T and U and returns a result of type R.

Functional Method:

```
R apply(T t, U u);
```

Example:

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(5, 4));  // 20
```

6. BinaryOperator<T>

- A **special case** of BiFunction<T, T, T>, returns the same type as input.

```
BinaryOperator<Integer> add = (a, b) -> a + b;
System.out.println(add.apply(2, 3)); // 5
```

7. UnaryOperator<T>

- A **special case** of Function<T, T> — one input, one output of same type.

```
UnaryOperator<String> toUpper = s -> s.toUpperCase();
System.out.println(toUpper.apply("hello")); // HELLO
```

6.2.4 Practice Problems with Solutions – Java 8 Functional Interfaces

1. Predicate<T>

Problem:

Filter out all even numbers from a list of integers.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
```

```
public class PredicateExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 3, 7, 6);
        Predicate<Integer> isEven = n -> n % 2 == 0;

        List<Integer> evenNumbers = numbers.stream()
                                           .filter(isEven)
                                           .collect(Collectors.toList());

        System.out.println(evenNumbers);  // Output: [2, 8, 6]
    }
}
```

2. Function<T, R>

Problem:

Convert a list of strings into their lengths.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class FunctionExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Java", "Python", "Go");

        Function<String, Integer> strLength = s -> s.length();

        List<Integer> lengths = names.stream()
                                     .map(strLength)
                                     .collect(Collectors.toList());

        System.out.println(lengths);  // Output: [4, 6, 2]
    }
}
```

3. Consumer<T>

Problem:

Print each string in uppercase.

Solution:

```
import java.util.*;
import java.util.function.*;

public class ConsumerExample {
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("apple", "banana", "mango");
```

```
        Consumer<String> printUpper = s -> System.out.println(s.toUpperCase());

        fruits.forEach(printUpper);
        // Output: APPLE BANANA MANGO
    }
}
```

4. Supplier<T>

Problem:

Generate and print 5 random double values.

Solution:

```
import java.util.function.*;
import java.util.stream.*;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomSupplier = () -> Math.random();

        List<Double> randomNumbers = Stream.generate(randomSupplier)
                                           .limit(5)
                                           .collect(Collectors.toList());

        System.out.println(randomNumbers);
    }
}
```

5. BiFunction<T, U, R>

Problem:

Create a full name from first and last name.

Solution:

```
import java.util.function.*;

public class BiFunctionExample {
    public static void main(String[] args) {
        BiFunction<String, String, String> fullName =
            (first, last) -> first + " " + last;

        System.out.println(fullName.apply("John", "Doe"));  // John Doe
    }
}
```

6. BinaryOperator<T>

Problem:

Add two integers.

Solution:

```
import java.util.function.*;

public class BinaryOperatorExample {
    public static void main(String[] args) {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(10, 15));  // Output: 25
    }
}
```

7. UnaryOperator<T>

Problem:

Add 10 to each number in a list.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class UnaryOperatorExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
        UnaryOperator<Integer> addTen = n -> n + 10;

        List<Integer> updated = numbers.stream()
                                       .map(addTen)
                                       .collect(Collectors.toList());

        System.out.println(updated);  // Output: [11, 12, 13, 14]
    }
}
```

6.2.5 Java 8 Stream API – Complete Guide

1. What is the Stream API?

The **Stream API** allows you to process **collections** (like List, Set, etc.) in a **functional style**.
It provides a high-level abstraction for processing sequences of elements with operations like filtering, mapping, sorting, and collecting.

Think of it like a **conveyor belt** — elements flow through a pipeline of operations.

2. Key Features

- Works with **Collections** and **arrays**
- Supports **lazy** and **parallel** operations
- Allows **pipelining** of multiple operations

- Promotes **declarative programming**

3. Stream Pipeline Structure

Collection -> Stream -> Intermediate Operations -> Terminal Operation -> Result

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.stream()
     .filter(s -> s.startsWith("A"))
     .map(String::toUpperCase)
     .forEach(System.out::println);  // Output: ALICE
```

4. Stream Creation

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
Stream<String> stream2 = Stream.of("x", "y", "z");
```

5. Intermediate Operations (returns Stream)

METHOD	DESCRIPTION
.FILTER()	Filters elements using a predicate
.MAP()	Transforms each element
.SORTED()	Sorts the elements
.DISTINCT()	Removes duplicates
.LIMIT(N)	Limits the result to n elements
.SKIP(N)	Skips the first n elements
.PEEK()	Debug stream content during processing

6. Terminal Operations (returns a result or side-effect)

METHOD	DESCRIPTION
.FOREACH()	Performs an action for each element
.COLLECT()	Converts to List, Set, Map, etc.
.COUNT()	Counts number of elements
.REDUCE()	Reduces elements to a single value
.ANYMATCH()	Checks if any element matches criteria
.ALLMATCH()	Checks if all elements match
.NONEMATCH()	Checks if no elements match
.FINDFIRST()	Gets the first element (Optional)
.FINDANY()	Gets any one element (Optional)

7. Common Use Cases with Examples

```
Filter and Print Names Starting with 'A'
names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);
Convert List of Strings to Uppercase
List<String> upper = names.stream()
                          .map(String::toUpperCase)
                          .collect(Collectors.toList());
Find Length of Each String
List<Integer> lengths = names.stream()
                             .map(String::length)
                             .collect(Collectors.toList());
Sum of Integers using reduce
List<Integer> nums = Arrays.asList(1, 2, 3, 4);
int sum = nums.stream()
              .reduce(0, (a, b) -> a + b);
System.out.println(sum); // 10
Count Unique Elements
long count = nums.stream()
                 .distinct()
                 .count();
Sort Strings by Length
List<String> sorted = names.stream()
                           .sorted(Comparator.comparing(String::length))
                           .collect(Collectors.toList());
```

8. Collecting Results

Use Collectors utility class for collecting stream data:

```
import java.util.stream.Collectors;

List<String> result = names.stream()
                           .filter(n -> n.length() > 3)
                           .collect(Collectors.toList());
```

6.2.6 Java 8: Default and Static Methods in Interfaces

Why Were These Introduced?

Before Java 8, interfaces could only have **abstract methods** — meaning all implementing classes had to provide their own definitions.

This made it hard to:

- **Add new methods** to interfaces without breaking existing code.
- Provide **shared behavior** among multiple classes.

Java 8 introduced default and static methods to solve these problems.

1. Default Methods

What is a Default Method?

A method in an interface that has a **default implementation** — introduced using the default keyword.

Syntax:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting...");
    }
}
```

Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car implements Vehicle {
    // Inherits default method unless overridden
}

public class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();  // Output: Vehicle starting...
    }
}
```

2. Static Methods

What is a Static Method in Interface?

Static methods belong to the **interface itself**, not to instances. They are called using the **interface name**.

Syntax:

```
interface Utility {
    static void log(String msg) {
        System.out.println("LOG: " + msg);
    }
}
```

Example:

```
public class Test {
    public static void main(String[] args) {
        Utility.log("Running app..."); // Output: LOG: Running app...
    }
}
```

Default vs Static

FEATURE	DEFAULT METHOD	STATIC METHOD
ACCESSED VIA	Object/instance	Interface name
INHERITED?	Yes (can be overridden)	No (not inherited)
USE CASE	Provide shared implementation	Provide utility/helper methods

Handling Multiple Inheritance Conflicts

If a class implements **two interfaces** with the same default method, the class **must override** that method to avoid conflict.

Example:

```
interface A {
    default void show() {
        System.out.println("From A");
    }
}

interface B {
    default void show() {
        System.out.println("From B");
    }
}

class C implements A, B {
    public void show() {
        A.super.show();  // or B.super.show();
    }
}
```

Real-World Analogy

- default method → Like a **default setting** in a mobile app — you can override it if needed.
- static method → Like a **utility tool** available in the app's settings — always there, same for all users.

Best Practices

- Use default methods **only** when adding new behavior to existing interfaces to maintain backward compatibility.
- Keep interfaces **clean and focused** — avoid overusing default/static for logic-heavy methods.

6.2.7 Java 8: Method Reference

What is a Method Reference?

A **method reference** is a **shorthand syntax** for a **lambda expression** that simply calls an existing method.

In other words:

If a lambda looks like this:

```
s -> System.out.println(s)
```

You can simplify it using a method reference:

```
System.out::println
```

Syntax Forms

TYPE	SYNTAX EXAMPLE	USED FOR
STATIC METHOD	ClassName::staticMethod	Math::max, Integer::parseInt
INSTANCE METHOD (OF OBJECT)	obj::instanceMethod	System.out::println
INSTANCE METHOD (OF TYPE)	ClassName::instanceMethod	String::length, String::toUpperCase
CONSTRUCTOR REFERENCE	ClassName::new	ArrayList::new, Employee::new

1. Reference to a Static Method

Lambda:

```
Function<Integer, String> func = n -> String.valueOf(n);
```

Method Reference:

```
Function<Integer, String> func = String::valueOf;
```

2. Reference to an Instance Method of a Particular Object

Lambda:

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

Method Reference:

```
Consumer<String> printer = System.out::println;
```

3. Reference to an Instance Method of an Arbitrary Object of a Particular Type

Lambda:

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

Method Reference:

```
Function<String, Integer> lengthFunc = String::length;
```

This is **very common** when working with streams:

```
List<String> names = Arrays.asList("Java", "Python", "C");
```

names.stream().map(String::toUpperCase).forEach(System.out::println);

4. Reference to a Constructor

Lambda:

```
Supplier<List<String>> listSupplier = () -> new ArrayList<>();
```

Method Reference:

```
Supplier<List<String>> listSupplier = ArrayList::new;
```

Also supports parameterized constructors:

```
Function<String, StringBuilder> sbFunc = StringBuilder::new;
System.out.println(sbFunc.apply("Hello"));  // Output: Hello
```

Real-World Example

Problem:

Convert a list of strings to uppercase and print them.

Using Lambda:

```
names.stream()
     .map(s -> s.toUpperCase())
     .forEach(s -> System.out.println(s));
Using Method References:
names.stream()
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

When to Use Method Reference?

Use it **only when**:

- The lambda calls a method directly
- It improves **readability** and **brevity**

Dont

This won't work:

```
Function<String, String> f = s -> s.concat("!");
```

You **cannot** write:

```
Function<String, String> f = String::concat;
```

Unless you already know that the second argument will be provided later. So be mindful about method **signatures**.

6.2.8 Java 8: Optional Class – A Guide to Avoid NullPointerException

What is Optional<T>?

Optional is a **container object** which may or may not contain a non-null value.

Think of it as a **box**:

- It may contain a value.
- ❌ It may be empty.

It forces you to **explicitly check** whether a value is present — helping you write **null-safe code**.

Why Use Optional?

Before Java 8:

String name = getName();

```
if (name != null) {
   System.out.println(name.length());
}
```

With Optional:

```
Optional<String> name = getName();
name.ifPresent(n -> System.out.println(n.length()));
```

Creating Optionals

METHOD	DESCRIPTION
OPTIONAL.OF(VALUE)	Creates Optional with a **non-null** value
OPTIONAL.OFNULLABLE(V)	Allows **null** or non-null
OPTIONAL.EMPTY()	Creates an **empty** Optional

Examples:

```
Optional<String> a = Optional.of("Java");          // Valid
Optional<String> b = Optional.ofNullable(null);    // Empty Optional
Optional<String> c = Optional.empty();             // Explicitly empty
```

Common Methods

METHOD	DESCRIPTION
ISPRESENT()	Returns true if value is present
IFPRESENT(CONSUMER)	Executes if value exists
GET()	Returns value, throws NoSuchElementException if empty (⚠ risky)
ORELSE(DEFAULT)	Returns value or default if empty
ORELSEGET(SUPPLIER)	Returns value or uses Supplier to compute default
ORELSETHROW()	Throws exception if value is empty
MAP(FUNCTION)	Transforms the value inside Optional
FILTER(PREDICATE)	Returns Optional if value passes filter
FLATMAP()	For nested Optionals

Examples

1. Using of and get

```
Optional<String> name = Optional.of("Alice");
System.out.println(name.get()); // Alice
```

2. Avoid null

```
Optional<String> name = Optional.ofNullable(null);
System.out.println(name.orElse("Default")); // Output: Default
```

3. ifPresent

```
name.ifPresent(n -> System.out.println("Hello " + n));
```

4. map and filter

```
Optional<String> name = Optional.of("Alice");

name.filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .ifPresent(System.out::println);  // Output: ALICE
```

Real-World Use Case: Avoiding Null in Service/DAO Return

```
public Optional<User> findUserById(int id) {
    User user = dao.find(id);
    return Optional.ofNullable(user);
}
```

In client code:

```
Optional<User> userOpt = service.findUserById(1);
userOpt.ifPresent(user -> System.out.println(user.getName()));
```

Don't Misuse Optional

❌ WRONG USAGE	BETTER ALTERNATIVE
AS METHOD PARAMETER	Use regular object (nullable)
IN CLASS FIELDS	Avoid — makes code noisy
FOR EVERY VALUE BLINDLY	Use only when null is expected

Summary

TASK	TRADITIONAL	WITH OPTIONAL
CHECK FOR NULL	if (obj != null)	optional.isPresent()
SAFE USE OF VALUE	if != null then	optional.ifPresent()
DEFAULT FALLBACK	if == null ? x	optional.orElse(x)
CHAINING OPERATIONS	Complex checks	optional.map().filter()

6.2.9 Java 8 Date and Time API (java.time)

Why a New API?

Old APIs like Date, Calendar, and SimpleDateFormat:

- Were **not thread-safe**
- Had **poor API design**
- Mixed **mutability** and **confusing behavior**
- Lacked **timezone support**

Java 8 solved this with a **cleaner, immutable, and thread-safe** Date-Time API inspired by Joda-Time.

Core Classes in java.time

Class	Purpose
LocalDate	Date without time (e.g., 2025-06-05)
LocalTime	Time without date (e.g., 10:15:30)
LocalDateTime	Date + Time (no timezone)
ZonedDateTime	Date + Time + Timezone
Period	Difference between dates (in years, months, days)
Duration	Difference between times (in seconds, nanos)
DateTimeFormatter	Formatting and parsing dates and times

1. LocalDate, LocalTime, LocalDateTime

LocalDate

```
LocalDate date = LocalDate.now(); // today's date
LocalDate dob = LocalDate.of(1995, 12, 15);
System.out.println(dob.getYear());  // 1995
System.out.println(dob.plusDays(5)); // 1995-12-20
```

LocalTime

```
LocalTime time = LocalTime.now();     // e.g., 14:23:45
LocalTime specific = LocalTime.of(9, 30);
System.out.println(specific.plusHours(2)); // 11:30
```

LocalDateTime

```
LocalDateTime dateTime = LocalDateTime.now();
System.out.println(dateTime);  // e.g., 2025-06-05T14:23:45
```

2. ZonedDateTime and ZoneId

Handle timezones accurately:

```
ZonedDateTime zoned = ZonedDateTime.now();
System.out.println(zoned);  // Includes offset and zone

ZoneId zone = ZoneId.of("Asia/Kolkata");
ZonedDateTime istTime = ZonedDateTime.now(zone);
System.out.println(istTime);
```

3. Period and Duration

Period (for LocalDate)

```
LocalDate start = LocalDate.of(2020, 1, 1);
LocalDate end = LocalDate.now();
Period p = Period.between(start, end);
System.out.println(p.getYears() + " years " + p.getMonths() + " months");
```

Duration (for LocalTime)

```
LocalTime t1 = LocalTime.of(10, 0);
LocalTime t2 = LocalTime.of(12, 30);
Duration d = Duration.between(t1, t2);
System.out.println(d.toMinutes());  // 150
```

4. Formatting and Parsing

Use DateTimeFormatter to convert date/time to/from Strings.

```
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

String formatted = now.format(formatter);
System.out.println(formatted);  // e.g., 05-06-2025 14:25

LocalDateTime parsed = LocalDateTime.parse("01-01-2020 10:00", formatter);
System.out.println(parsed);
```

5. Conversion with Old API

```
Date date = new Date();
Instant instant = date.toInstant();
LocalDateTime ldt = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
```

Summary of Improvements

Feature	Old API	New API (java.time)
Thread-safe	❌	✅
Immutable	❌	✅
Readable	❌ Complex API	Clear, fluent API
Timezones	Limited, error-prone	Full ZoneId, ZonedDateTime
Formatting	Confusing SimpleDateFormat	Powerful DateTimeFormatter

Real-World Use Case

Imagine a booking system:

- LocalDate for event date
- LocalTime for time slots
- ZonedDateTime for global user support
- Period to calculate membership duration
- DateTimeFormatter for displaying times cleanly

6.2.9 Java 8 Collectors (from java.util.stream.Collectors)

What Are Collectors?

Collectors are utility methods that transform a stream's elements into:

- **Collections** (List, Set, Map)
- **Summarized values** (sum, avg, count)
- **Grouped data**
- **Joined Strings**

They work with the Stream.collect() method.

List<String> names = list.stream().collect(Collectors.toList());

Commonly Used Collectors

COLLECTOR	DESCRIPTION
TOLIST()	Collect elements into a List
TOSET()	Collect elements into a Set
TOMAP(KEYMAPPER, VALUEMAPPER)	Collect elements into a Map
JOINING()	Concatenate strings
COUNTING()	Count number of elements
SUMMARIZINGINT() / SUMMARIZINGDOUBLE()	Returns count, sum, min, avg, max
GROUPINGBY(CLASSIFIER)	Group elements based on a property
PARTITIONINGBY(PREDICATE)	Split into true/false lists
MAPPING()	Map + Collect in nested collectors

Examples

1. toList()

```
List<String> names = Stream.of("A", "B", "C")
    .collect(Collectors.toList());
```

2. toSet()

```
Set<Integer> nums = Stream.of(1, 2, 2, 3)
    .collect(Collectors.toSet());  // Removes duplicates
```

3. toMap()

```
List<String> words = Arrays.asList("Java", "Python");

Map<String, Integer> wordLengths = words.stream()
    .collect(Collectors.toMap(w -> w, w -> w.length()));
If keys may duplicate, use merge function:
.collect(Collectors.toMap(w -> w, w -> 1, Integer::sum));
```

4. joining()

```
List<String> list = Arrays.asList("One", "Two", "Three");

String result = list.stream()
    .collect(Collectors.joining(", "));  // One, Two, Three
```

5. counting()

```
long count = list.stream()
    .collect(Collectors.counting());
```

6. summarizingInt()

```
IntSummaryStatistics stats = Stream.of(1, 2, 3, 4)
    .collect(Collectors.summarizingInt(i -> i));

System.out.println(stats.getAverage());  // 2.5
```

7. groupingBy()

```
class Student {
    String name;
    String dept;

    Student(String name, String dept) {
        this.name = name;
        this.dept = dept;
    }
}
List<Student> students = Arrays.asList(
    new Student("A", "CSE"),
    new Student("B", "ECE"),
    new Student("C", "CSE")
);

Map<String, List<Student>> grouped = students.stream()
    .collect(Collectors.groupingBy(s -> s.dept));
```

8. partitioningBy()

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Map<Boolean, List<Integer>> evenOdd = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));
```

9. mapping() (Nested collection transform)

```
Map<String, List<String>> deptNames = students.stream()
    .collect(Collectors.groupingBy(
        s -> s.dept,
        Collectors.mapping(s -> s.name, Collectors.toList())
    ));
```

Summary

TASK	COLLECTOR
CONVERT TO LIST	`toList()`
CONVERT TO SET	`toSet()`
CONVERT TO MAP	`toMap()`
CONCATENATE STRINGS	`joining()`
COUNT ELEMENTS	`counting()`
GET STATS (AVG, MIN, MAX)	`summarizingInt()`
GROUP BY FIELD	`groupingBy()`
PARTITION BY TRUE/FALSE	`partitioningBy()`
COLLECT & TRANSFORM	`mapping()`