Java APIs

Java Date and Time API (java.time)

Definition

The **Java Date and Time API** (java.time) is a modern API introduced in **Java 8** to handle **date, time, duration, and time zones** in a clear, immutable, and thread-safe way.

Why it is Needed / Purpose

Older APIs (java.util.Date, Calendar) had problems:

- Mutable (can change accidentally)
- Not thread-safe
- Confusing methods

java.time solves these by being:

- **Immutable**
- **Thread-safe**
- **Easy to read and use**

Key Packages & Classes

Main package: **java.time**

Important classes:

1. **LocalDate** – date only (YYYY-MM-DD)
2. **LocalTime** – time only (HH:MM:SS)
3. **LocalDateTime** – date + time
4. **ZonedDateTime** – date + time + timezone
5. **Period** – date-based difference
6. **Duration** – time-based difference
7. **DateTimeFormatter** – formatting & parsing

How It Works (Conceptually)

- You **get current date/time** using now()
- You **create custom dates** using of()
- You **format/parse** dates using DateTimeFormatter
- Objects are **immutable** → any change creates a new object

---

Real-Life Example (Easy Analogy)

- **LocalDate** → Birthdate (no time needed)
- **LocalTime** → Office opening time
- **LocalDateTime** → Exam date & time
- **ZonedDateTime** → Online meeting across countries

Technical Examples (Very Important)

LocalDate

```
LocalDate today = LocalDate.now();
LocalDate dob = LocalDate.of(2002, 5, 10);
```

LocalTime

```
LocalTime now = LocalTime.now();
LocalTime start = LocalTime.of(9, 30);
```

LocalDateTime

```
LocalDateTime current = LocalDateTime.now();
```

ZonedDateTime

```
ZonedDateTime indiaTime = ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));
```

Formatting Date & Time

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("dd-MM-yyyy");

String formattedDate = LocalDate.now().format(formatter);
```

Output example:

```
22-01-2026
```

Period vs Duration

Feature	Period	Duration
Based on	Date	Time
Used for	Years, months, days	Hours, minutes, seconds
Example	Age	Time difference

```
Period age = Period.between(dob, LocalDate.now());
Duration timeGap = Duration.between(start, now);
```

Advantages

- Clean and readable API
- Immutable & thread-safe
- Better timezone handling
- Recommended by Java standards

Disadvantages / Limitations

- Slight learning curve for beginners
- Older legacy code still uses Date & Calendar

Common Interview Questions (Cognizant Level)

Q1. Why was java.time introduced?
A. To fix problems of old Date & Calendar APIs.

Q2. Difference between LocalDate and LocalDateTime?
A. LocalDate = date only
LocalDateTime = date + time

Q3. Is java.time thread-safe? Why?
A. Yes, because all classes are immutable.

Q4. How to handle time zones in Java 8?
A. Using ZonedDateTime and ZoneId.

Q5. Which class is used for formatting dates?
A. DateTimeFormatter

One-Line Summary (Revision Ready)

Java Date and Time API (java.time) is a modern, immutable, and thread-safe way to handle date, time, and time zones in Java 8+.

Java Reflection API

Definition:

Java Reflection API allows a program to **inspect and manipulate classes, methods, fields, and constructors at runtime**, even if they are **private**.

Why it is Needed / Purpose

Normally, Java code is checked at **compile time**.

Reflection is needed when:

- Class names are **not known at compile time**
- Frameworks need **dynamic behavior**
- Tools need to **analyze classes at runtime**

Widely used in **Spring, Hibernate, JUnit**

Key Packages & Classes

Package: **java.lang.reflect**

Important classes:

1. **Class** – represents a loaded class
2. **Method** – represents class methods
3. **Field** – represents variables
4. **Constructor** – represents constructors
5. **Modifier** – access modifiers info

How It Works (Conceptual Flow)

1. JVM loads class
2. Reflection API gets **Class object**
3. Class object gives metadata (methods, fields, etc.)
4. JVM executes methods dynamically

Real-Life Example (Analogy)

- **Normal Java** → You know the machine and buttons beforehand
- **Reflection** → You open the machine and inspect **what buttons exist** and how they work at runtime

Technical Examples (Core Part)

Get Class Object (3 Ways)

```
Class<?> c1 = Student.class;
Class<?> c2 = obj.getClass();
Class<?> c3 = Class.forName("com.example.Student");
```

Access Methods

```
Method[] methods = c1.getDeclaredMethods();
for(Method m : methods) {
    System.out.println(m.getName());
}
```

Access Private Field

```
Field f = c1.getDeclaredField("name");
f.setAccessible(true);
f.set(obj, "Ali");
```

Invoke Method

```
Method m = c1.getDeclaredMethod("display");
m.invoke(obj);
```

Real-Time Usage in Frameworks

- **Spring** → Dependency Injection
- **Hibernate** → ORM mapping
- **JUnit** → Running test methods
- **Servlets** → Dynamic class loading

Advantages

- Powerful and flexible
- Enables frameworks
- Useful for debugging & testing tools

Disadvantages / Limitations

- Slower performance
- Breaks encapsulation
- Security risks if misused
- Harder to read & maintain

Common Interview Questions (Cognizant Level)

Q1. What is Reflection in Java?
A. Inspect and manipulate classes at runtime.
Q2. Why is Reflection considered powerful but dangerous?
A. It can access private members and affect security.
Q3. Which class is the entry point of Reflection API?
A. java.lang.Class
Q4. Is Reflection used at compile time or runtime?
A. Runtime.
Q5. Give one real-time example of Reflection.
A. Spring Dependency Injection.

When NOT to Use Reflection

- Normal business logic
- Performance-critical code
- Simple applications

One-Line Summary (Quick Revision)
Reflection API lets Java programs inspect and manipulate class details at runtime and is mainly used by frameworks.

Java Stream API

Definition:
Java Stream API (introduced in **Java 8**) is used to **process collections of data in a functional style**, such as filtering, mapping, and reducing data.

- A **Stream is not a data structure**, it is a **pipeline for data processing**.

Why it is Needed / Purpose

Before Java 8:

- Long loops
- More boilerplate code
- Hard to read

Stream API helps to:

- Write **clean and readable code**
- Perform **bulk operations**
- Support **parallel processing**

Key Components of Stream API

Stream processing has **3 parts**:

1. **Source** – Collection / Array

2. **Intermediate Operations** – Transform data
3. **Terminal Operation** – Produce result

How It Works (Flow Pattern)

Collection → Stream → Intermediate Ops → Terminal Op → Result
Example:
List → filter → map → collect

Real-Life Example (Easy Analogy)

Think of a **water pipe**:

- Water enters (source)
- Filters clean it (intermediate)
- Tap gives output (terminal)

Core Stream Operations

Common Intermediate Operations

- filter()
- map()
- sorted()
- distinct()
- limit()

Common Terminal Operations

- forEach()
- collect()
- count()
- reduce()
- findFirst()

Technical Examples (Very Important)

1. **Filter Example**

```
List<Integer> nums = List.of(1, 2, 3, 4, 5);

nums.stream()
    .filter(n -> n % 2 == 0)
    .forEach(System.out::println);
```

---

2. **Map Example**

```
nums.stream()
    .map(n -> n * n)
    .forEach(System.out::println);
```

---

3. **Collect Example**

```
List<Integer> evenNums =
    nums.stream()
        .filter(n -> n % 2 == 0)
        .collect(Collectors.toList());
```

Intermediate vs Terminal Operations

Feature	**Intermediate**	**Terminal**
Return type	Stream	Result
Lazy execution	Yes	No
Example	filter(), map()	collect(), forEach()

Parallel Stream

```
nums.parallelStream()
    .forEach(System.out::println);
```

- Uses **multiple threads** for faster processing (large data).

Advantages

- Less code
- More readable
- Supports parallelism
- Functional programming support

Disadvantages / Limitations

- Not good for small/simple tasks
- Debugging is harder
- Performance overhead in small collections

Common Interview Questions (Cognizant Level)

Q1. Is Stream a data structure?
A. No, it only processes data.

Q2. Difference between map() and filter()?
A. filter() selects elements
map() transforms elements

Q3. Can we reuse a Stream?
A. No, stream is single-use.

Q4. What is lazy evaluation?
A. Operations execute only when terminal operation is called.

Q5. Difference between stream() and parallelStream()?
A. stream() → sequential
parallelStream() → parallel

Real-Time Usage

- Filtering active users
- Processing logs
- Data analytics
- Reporting systems

One-Line Summary (Quick Revision)

Stream API processes collections using functional-style operations with clean, readable, and efficient code.
