Date and Time In Java

- The **`Date` class** represents a specific **instant in time**, with millisecond precision.
- It is part of the **old date-time API**.
- Most of its methods are **deprecated** (replaced by `java.time` package in Java 8+).

Common Constructors

```
Date();  // Creates a Date object with the current date and time
Date(long milliseconds);  // Creates a Date object from milliseconds since
Jan 1, 1970
```

Common Methods

```
getTime()       // Returns time in milliseconds since Jan 1, 1970
after(Date d)   // Checks if current date is after given date
before(Date d)  // Checks if current date is before given date
compareTo(Date d) // Compares two dates
toString()      // Converts date to String
```

Example:

```
import java.util.Date;

public class DateExample {
    public static void main(String[] args) {
        Date d1 = new Date(); // Current date and time
        System.out.println("Current Date: " + d1);

        Date d2 = new Date(1000000000L);
        System.out.println("Specific Date: " + d2);

        System.out.println("Is d1 after d2? " + d1.after(d2));
    }
}
```

Calendar and GregorianCalendar in Java

Java provides classes to work with **date and time** in a more structured way than the old `Date` class.

1. `Calendar` Class

- An **abstract class** in `java.util` package.
- Provides methods to manipulate dates and times (e.g., add days, months, years).
- You cannot create `Calendar` objects directly (because it is abstract).
- Instead, you use the `getInstance()` method.

Important Points

- Stores information like **year, month, day, hour, minute, second**.
- Months are **zero-based** → January = 0, February = 1, ..., December = 11.
- Days are **1-based** → Sunday = 1, Monday = 2, ..., Saturday = 7.

Common Methods

```
get(int field)   // Returns the value of the given field (YEAR, MONTH,
DAY_OF_MONTH, etc.)
set(int field, int value)  // Sets the given field
add(int field, int amount) // Adds/subtracts from a field
getTime()        // Returns Date object
getInstance()    // Gets a Calendar instance based on system default time
zone
```

Example:

```
import java.util.*;

public class CalendarExample {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();

        System.out.println("Year: " + cal.get(Calendar.YEAR));
        System.out.println("Month: " + (cal.get(Calendar.MONTH) + 1)); //
+1 because months are 0-based
        System.out.println("Day: " + cal.get(Calendar.DAY_OF_MONTH));
        System.out.println("Hour: " + cal.get(Calendar.HOUR));
        System.out.println("Minute: " + cal.get(Calendar.MINUTE));
        System.out.println("Second: " + cal.get(Calendar.SECOND));

        // Add 10 days
        cal.add(Calendar.DAY_OF_MONTH, 10);
        System.out.println("After 10 days: " + cal.getTime());
    }
}
```

2. `GregorianCalendar` Class

- A **concrete subclass** of `Calendar`.
- Implements the **Gregorian calendar system** (the standard calendar used worldwide).
- Supports **BC (Before Christ)** and **AD (Anno Domini)** eras.

Constructors

```
GregorianCalendar();   // Current date and time
GregorianCalendar(int year, int month, int day);
GregorianCalendar(int year, int month, int day, int hour, int minute, int
second);
```

Extra Features

- `isLeapYear(int year)` → Checks if a year is a leap year.
- Provides more control for **historical calendar dates**.

Example:

```
import java.util.*;

public class GregorianCalendarExample {
    public static void main(String[] args) {
        GregorianCalendar gcal = new GregorianCalendar();
```

```
        System.out.println("Date: " + gcal.getTime());
        System.out.println("Year: " + gcal.get(Calendar.YEAR));
        System.out.println("Month: " + (gcal.get(Calendar.MONTH) + 1));
        System.out.println("Day: " + gcal.get(Calendar.DAY_OF_MONTH));

        // Leap year check
        int year = 2024;
        System.out.println(year + " is leap year? " +
gcal.isLeapYear(year));
    }
}
```

3. Difference Between `Calendar` and `GregorianCalendar`

Feature	`Calendar`	`GregorianCalendar`
Type	Abstract class	Concrete subclass
Usage	General-purpose calendar API	Specific implementation of the Gregorian system
Instantiation	Cannot directly (use `getInstance()`)	Can create objects directly
Leap Year Support	Not directly available	`isLeapYear(int year)` method
Eras (BC/AD)	Supported	Fully supported

`java.time` API in Java

The **`java.time`** package was introduced in **Java 8** to replace the old `Date` and `Calendar` classes.
It provides a **modern, immutable, and thread-safe** date-time API.

1. Key Classes in `java.time`

Class	Description
`LocalDate`	Represents **date only** (Year, Month, Day) without time and timezone
`LocalTime`	Represents **time only** (Hour, Minute, Second, Nano) without date and timezone
`LocalDateTime`	Represents **date + time** (no timezone)
`ZonedDateTime`	Represents **date + time + timezone**
`Instant`	Represents a **point in time (timestamp)** in UTC
`Period`	Represents the difference between **dates** (years, months, days)
`Duration`	Represents the difference between **times** (hours, minutes, seconds)

Class	Description
`Clock`	Represents the system clock

Example: Using `LocalDate`, `LocalTime`, `LocalDateTime`

```
import java.time.*;

public class TimeApiExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.now();   // Current date
        LocalTime time = LocalTime.now();   // Current time
        LocalDateTime dateTime = LocalDateTime.now(); // Current date-time
        ZonedDateTime zoned = ZonedDateTime.now(); // With time zone

        System.out.println("Date: " + date);
        System.out.println("Time: " + time);
        System.out.println("DateTime: " + dateTime);
        System.out.println("ZonedDateTime: " + zoned);
    }
}
```

2. Performing Operations

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plusDays(1);  // Add 1 day
LocalDate nextMonth = today.plusMonths(1); // Add 1 month
LocalDate previousYear = today.minusYears(1); // Subtract 1 year

System.out.println("Tomorrow: " + tomorrow);
System.out.println("Next Month: " + nextMonth);
System.out.println("Previous Year: " + previousYear);
```

3. Measuring Time: Period and Duration

```
import java.time.*;

public class PeriodDurationExample {
    public static void main(String[] args) {
        LocalDate start = LocalDate.of(2020, 1, 1);
        LocalDate end = LocalDate.now();
        Period period = Period.between(start, end);
        System.out.println("Years: " + period.getYears() + ", Months: " +
period.getMonths() + ", Days: " + period.getDays());

        LocalTime t1 = LocalTime.of(10, 0);
        LocalTime t2 = LocalTime.of(12, 30);
        Duration duration = Duration.between(t1, t2);
        System.out.println("Duration in minutes: " + duration.toMinutes());
    }
}
```

DateTimeFormatter

- The **`DateTimeFormatter`** class is used to **format** and **parse** date-time objects.
- Part of `java.time.format` package.
- Works with `LocalDate`, `LocalTime`, `LocalDateTime`, etc.

1. Predefined Formatters

```
import java.time.*;
import java.time.format.DateTimeFormatter;

public class FormatterExample1 {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();

        System.out.println("ISO_DATE: " +
now.format(DateTimeFormatter.ISO_DATE));
        System.out.println("ISO_DATE_TIME: " +
now.format(DateTimeFormatter.ISO_DATE_TIME));
    }
}
```

2. Custom Patterns

Symbols used:

- `dd` → Day (2 digits)
- `MM` → Month (2 digits)
- `yyyy` → Year (4 digits)
- `HH` → Hour (24h)
- `hh` → Hour (12h)
- `mm` → Minutes
- `ss` → Seconds
- `a` → AM/PM

```
import java.time.*;
import java.time.format.DateTimeFormatter;

public class FormatterExample2 {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-
yyyy HH:mm:ss a");
        String formattedDate = now.format(formatter);

        System.out.println("Formatted Date-Time: " + formattedDate);
    }
}
```

3. Parsing Strings to Date-Time

```
String dateStr = "22-09-2025 14:30:00";
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");
LocalDateTime dateTime = LocalDateTime.parse(dateStr, formatter);

System.out.println("Parsed Date-Time: " + dateTime);
```

Difference Between SimpleDateFormat (Old) vs DateTimeFormatter (New)

Feature	`SimpleDateFormat` (Old)	`DateTimeFormatter` (New)
Package	`java.text`	`java.time.format`

Feature	`SimpleDateFormat` (Old)	`DateTimeFormatter` (New)
Thread Safety	Not thread-safe	Thread-safe
Mutability	Mutable (unsafe)	Immutable (safe)
Usage	Legacy	Modern (Java 8+)

Generating Your Own Docs in Java (Javadoc)

Java provides a tool called **`javadoc`** to automatically generate documentation from your source code.
It reads the **comments** written in a **special format** (called *Javadoc comments*) and produces **HTML documentation**.

1. What is Javadoc?

- A **documentation generator** included in the JDK.
- Creates HTML docs directly from your Java source code.
- Uses `/** ... */` comments (Javadoc comments).
- Useful for creating documentation of **classes, methods, variables, and packages**.

2. Javadoc Comments

- Written using `/** ... */` (not `/* ... */` or `// ...`).
- Can include **tags** like `@author`, `@version`, `@param`, `@return`, etc.

Example: Javadoc Comment

```
/**
 * This class represents a simple Calculator.
 * It supports basic arithmetic operations.
 *
 * @author Neeraj
 * @version 1.0
 */
public class Calculator {

    /**
     * Adds two integers.
     * @param a first number
     * @param b second number
     * @return sum of a and b
     */
    public int add(int a, int b) {
        return a + b;
    }

    /**
     * Subtracts two integers.
     * @param a first number
     * @param b second number
     * @return difference of a and b
     */
```

```
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

3. Useful Javadoc Tags

Tag	Usage
`@author`	Defines the author of the class/file
`@version`	Defines the version of the class/file
`@param`	Describes a parameter of a method
`@return`	Describes what a method returns
`@throws` or `@exception`	Describes exceptions thrown
`@see`	Links to another class/method
`@since`	Version of JDK since this code exists

4. Generating Documentation

Step 1: Write Javadoc comments in your code.

Step 2: Compile documentation using `javadoc` tool.

```
javadoc -d docs Calculator.java
```

- `-d docs` → Stores generated docs in the folder `docs`.
- `Calculator.java` → Source file.

After running, open `docs/index.html` in a browser → You'll see **beautiful documentation**.

5. Documenting a Package

If your code is inside a **package**, Javadoc can generate package-level documentation too.

Example

File: `mypackage/MyClass.java`

```
package mypackage;

/**
 * A simple class inside mypackage.
 * @author Neeraj
 */
public class MyClass {
    /**
     * Prints a message.
     */
```

```
    public void showMessage() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

Generate docs for the package:

```
javadoc -d docs mypackage/*.java
```

Now `docs` folder will contain documentation for the entire **package**.

Javadoc Tags in Java

The **`javadoc` tool** generates documentation in **HTML format** from specially formatted comments in source code.
To make documentation useful, we use **tags**.

1. Javadoc Comment Syntax

- Written using `/** ... */` (not `//` or `/* ... */`).
- Placed **above classes, methods, constructors, variables, or packages**.

```
/**
 * Description of the class/method/field.
 * @tagName tag-description
 */
```

2. Tags for Documenting a Class / Interface

Tag	Usage Example
`@author`	`@author Neeraj`
`@version`	`@version 1.0`
`@since`	`@since JDK 1.8`
`@see`	`@see AnotherClass`
`@deprecated`	Marks class as deprecated → `@deprecated use NewClass instead`

Example:

```
/**
 * A simple Calculator class for arithmetic operations.
 *
 * @author Neeraj
 * @version 1.0
 * @since JDK 1.8
 */
public class Calculator {
    ...
}
```

3. Tags for Documenting a Method / Constructor

Tag	Usage Example
`@param`	Describes a parameter → `@param a first number`
`@return`	Describes return value → `@return sum of two numbers`
`@throws` or `@exception`	Describes exceptions → `@throws ArithmeticException if division by zero occurs`
`@see`	Reference → `@see #subtract(int, int)`
`@deprecated`	Marks method as deprecated → `@deprecated use addNumbers() instead`

Example:

```
/**
 * Adds two integers.
 *
 * @param a first integer
 * @param b second integer
 * @return sum of a and b
 */
public int add(int a, int b) {
    return a + b;
}

/**
 * Divides two integers.
 *
 * @param a numerator
 * @param b denominator
 * @return result of division
 * @throws ArithmeticException if b is zero
 */
public int divide(int a, int b) throws ArithmeticException {
    return a / b;
}
```

4. Miscellaneous Tags

Tag	Usage
`{@code ...}`	Displays code in monospace font → `{@code int x = 10;}`
`{@link ...}`	Inline link → `{@link Calculator#add(int, int)}`
`{@literal ...}`	Shows text literally without parsing HTML
`{@docRoot}`	Relative path to root of docs
`{@inheritDoc}`	Inherits documentation from parent class/interface

5. Generating Docs

Use command:

```
javadoc -d docs Calculator.java
```

Docs will be generated in the `docs/` folder → open `index.html`.