Full stack web development

By :- Prof. Praveen Tomar

Java Date and Time:

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time.
Following classes are important for dealing with date in Java.

Java 8 Date/Time API
Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

- java.time.LocalDate class
- java.time.LocalTime class
- java.time.LocalDateTime class
- java.time.MonthDay class
- java.time.OffsetTime class
- java.time.OffsetDateTime class
- java.time.Clock class
- java.time.ZonedDateTime class
- java.time.ZoneId class
- java.time.ZoneOffset class
- java.time.Year class

Classical Date/Time API

- But classical or old Java Date API is also useful. Let's see the list of classical Date and Time classes.
- java.util.Date class
- java.sql.Date class
- java.util.Calendar class
- java.util.GregorianCalendar class
- java.util.TimeZone class
- java.sql.Time class
- java.sql.Timestamp class

Formatting Date and Time

We can format date and time in Java by using the following classes:

- java.text.DateFormat class
- java.text.SimpleDateFormat class

Java Date and Time APIs:

Java provide the date and time functionality with the help of two packages java.time and java.util. The package java.time is introduced in Java 8, and the newly introduced classes tries to overcome the shortcomings of the legacy java.util.Date and java.util.Calendar classes.

Classical Date Time API Classes:

- **Java.lang.System:** The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.
- **java.util.Date:** It is used to show specific instant of time, with unit of millisecond.
- **java.util.Calendar:** It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.
- **java.text.SimpleDateFormat:** It is a class that is used to format and parse the dates in a predefined manner or user defined pattern.
- **java.util.TimeZone:** It represents a time zone offset, and also figures out daylight savings.

Drawbacks of existing Date/Time API's

Thread safety:

The existing classes such as Date and Calendar does not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care by developers. The new Date and Time APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.

Bad API designing:

The classic Date and Calendar APIs does not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provides number of different methods for performing operations regarding date, time, duration and periods.

Difficult time zone handling:

To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.

New Date Time API in Java 8

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

- **java.time.LocalDate:** It represents a year-month-day in the ISO calendar and is useful for representing a date without a time. It can be used to represent a date only information such as a birth date or wedding date.
- **java.time.LocalTime:** It deals in time only. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.
- **java.time.LocalDateTime:** It handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.
- **java.time.ZonedDateTime:** It combines the LocalDateTime class with the zone information given in ZoneId class. It represent a complete date time stamp along with timezone information.
- **java.time.OffsetTime:** It handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

- **java.time.OffsetDateTime:** It handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.
- **java.time.Clock :** It provides access to the current instant, date and time in any given time-zone. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.
- **java.time.Instant :** It represents the start of a nanosecond on the timeline (since EPOCH) and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.
- **java.time.Duration :** Difference between two instants and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.
- **java.time.Period :** It is used to define the difference between dates in date-based values (years, months, days).
- **java.time.ZoneId :** It states a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.
- **java.time.ZoneOffset :** It describe a time zone offset from Greenwich/UTC time.
- **java.time.format.DateTimeFormatter :** It comes up with various predefined formatter, or we can define our own. It has parse() or format() method for parsing and formatting the date time values.

Java Date and Time

Java Dates

Java does not have a built-in Date class, but we can import the `java.time` package to work with the date and time API. The package includes many date and time classes. For example:

Class	Description
LocalDate	Represents a date (year, month, day (yyyy-MM-dd))
LocalTime	Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
LocalDateTime	Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
DateTimeFormatter	Formatter for displaying and parsing date-time objects

Display Current Date

To display the current date, import the `java.time.LocalDate` class, and use its `now()` method:

```
import java.time.LocalDate;  // import the LocalDate class


public class Main {
  public static void main(String[] args) {
    LocalDate myObj = LocalDate.now();  // Create a date object
    System.out.println(myObj);  // Display the current date
  }
}
```

The output will be:
2025-01-06

Display Current Time

To display the current time (hour, minute, second, and nanoseconds), import the `java.time.LocalTime` class, and use its `now()` method:

```
import java.time.LocalTime;  // import the LocalTime class

public class Main {
  public static void main(String[] args) {
    LocalTime myObj = LocalTime.now();
    System.out.println(myObj);
  }
}

// Note: This example displays the server's local time, which may differ from your local time.
```

```
Output:-
                7:35:46.395597
```

Display Current Date and Time

To display the current date and time, import the `java.time.LocalDateTime` class, and use its `now()` method:

```
import java.time.LocalDateTime;  // import the LocalDateTime class

public class Main {
  public static void main(String[] args) {
    LocalDateTime myObj = LocalDateTime.now();
    System.out.println(myObj);
  }
}
```

Output:-

```
2025-01-05T07:50:39.648619
```

```
// Note: This example displays the server's local time, which may differ from your local time.
```

Formatting Date and Time:-

The "T" in the example above is used to separate the date from the time. You can use the `DateTimeFormatter` class with the `ofPattern()` method in the same package to format or parse date-time objects.

The following example will remove both the "T" and nanoseconds from the date-time:

```
import java.time.LocalDateTime;  // Import the LocalDateTime class
import java.time.format.DateTimeFormatter;  // Import the DateTimeFormatter class

public class Main {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After formatting: " + formattedDate);
  }
}
```

```
Output:-
Before Formatting:
2025-01-05T07:55:57.053598
After Formatting: 05-01-2025 07:55:57
```

```
// Note: This example displays the server's local time, which may differ from your local time.
```

The `ofPattern()` method accepts all sorts of values, if you want to display the date and time in a different format. For example:

Value	Example
yyyy-MM-dd	"1988-09-29"
dd/MM/yyyy	"29/09/1988"
dd-MMM-yyyy	"29-Sep-1988"
E, MMM dd yyyy	"Thu, Sep 29 1988"

```
import java.time.LocalDateTime;  // Import the LocalDateTime class
import java.time.format.DateTimeFormatter;  // Import the DateTimeFormatter class

public class Main {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After formatting: " + formattedDate);
  }
}
```

Output:-
Before Formatting: 2025-01-05T08:01:06.585315
After Formatting: 05-01-2025 08:01:06

```
import java.time.LocalDateTime;  // Import the LocalDateTime class
import java.time.format.DateTimeFormatter;  // Import the DateTimeFormatter class


public class Main {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before Formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After Formatting: " + formattedDate);
  }
}
```

Output:-
Before Formatting: 2025-01-05T08:05:05.436484
After Formatting: 05/01/2025 08:05:05

```
import java.time.LocalDateTime;  // Import the LocalDateTime class
import java.time.format.DateTimeFormatter;  // Import the DateTimeFormatter class

public class Main {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before Formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MMM-yyyy HH:mm:ss");

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After Formatting: " + formattedDate);
  }
}
```

Output:-
Before Formatting: 2025-01-05T08:07:26.281088
After Formatting: 05-Jan-2025 08:07:26

```
import java.time.LocalDateTime;  // Import the LocalDateTime class
import java.time.format.DateTimeFormatter;  // Import the DateTimeFormatter class

public class Main {
  public static void main(String[] args) {
    LocalDateTime myDateObj = LocalDateTime.now();
    System.out.println("Before Formatting: " + myDateObj);
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("E, MMM dd yyyy HH:mm:ss");

    String formattedDate = myDateObj.format(myFormatObj);
    System.out.println("After Formatting: " + formattedDate);
  }
}
```

Output:-
Before Formatting: 2025-01-05T08:08:32.480060
After Formatting: Sun, Jan 05 2025 08:08:32

java.time.MonthDay Class in Java

- Java is the most popular programming language and widely used programming language. Java is used in all kinds of applications like mobile applications, desktop applications, web applications. The java. time.MonthDay class represents a combination of month and day of the month, and it is immutable. java.time is a package used to work with the current date and time API. All the methods of this class are discussed below in the tabular format.

Method	Description
adjustInto(Temporal temporal)	Adjusts the specified temporal object to having this month-day.
atYear(int year)	Combines this month-day with a year to create a LocalDate.
compareTo(MonthDay other)	Compares this month-day to another month-day.
format(DateTimeFormatter formatter)	Formats this month-day using the specified formatter.
getDayOfMonth()	Gets the day-of-month field.
getMonth()	Gets the month-of-year field using the Month enum.
getMonthValue()	Gets the month-of-year field from 1 to 12.
hashCode()	A hash code for this month-day.
isAfter(MonthDay other)	Checks if this month-day is after the specified month-day.

now()	Obtains the current month-day from the system clock in the default time-zone.
now(Clock clock)	Obtains the current month-day from the specified clock.
of(int month, int dayOfMonth)	Obtains an instance of MonthDay.
query(TemporalQuery<R> query)	Queries this month-day using the specified query.
range(TemporalField field)	Gets the range of valid values for the specified field.
toString()	Outputs this month-day as a String, such as –12-03.
with(Month month)	Returns a copy of this MonthDay with the month-of-year altered.
withDayOfMonth(int dayOfMonth)	Returns a copy of this MonthDay with the day-of-month altered.
withMonth(int month)	Returns a copy of this MonthDay with the month-of-year altered.

Implementation: Let us now discuss a few of the methods of this class

- Import classes and package java.time.
- Now use method such as MonthDay.of() or any other method and store instance of MonthDay.
- Display the stored value in a variable.

```
// Java Program to illustrate MonthDay Class

// Importing Month and MonthDay classes
// from java.time package
import java.time.Month;
import java.time.MonthDay;

// Main Class
public class GFG {

// Main driver method
public static void main(String[] args)
{
// Creating an object of MonthDay class by
// storing instance of MonthDay by
```

Output--03-14

```
// Java Program to illustrate MonthDay Class

// importing MonthDay class from java.time
import java.time.MonthDay;

// Main Class
public class GFG {

// Main driver method
public static void main(String[] args)
{
// Store an instance of MonthDay
// from a text i.e --03-14
MonthDay monthday = MonthDay.parse("--03-14");

// Display the month using instance of class
System.out.println(monthday.getMonth());
```

Output MARCH

Java OffsetDateTime Class

The OffsetDateTime class in Java is a crucial component of the java.time package introduced in Java 8. It represents a date-time with an offset from UTC/Greenwich, which is a crucial aspect for handling time zones and daylight saving time changes. it combines a date and time with an offset to accurately handle instances where time zones vary or when daylight saving time applies.

Features of OffsetDateTime Class:

- **Immutability:** Instances of OffsetDateTime are immutable, meaning once created, their values cannot be changed.
- **Precision:** It stores date, time, and offset with nanosecond precision, allowing for precise calculations.
- **Offset:** It includes the offset from UTC, which is crucial for handling time zone differences
- **Compatibility:** OffsetDateTime is designed to be compatible with other date-time classes in the java.time package, facilitating seamless conversion and interoperability.
- **Range:** It covers the range of dates from January 1, 0001 to December 31, 9999, and supports all valid offsets.

Differences from other Date-Time Classes:

- **LocalDateTime:** LocalDateTime does not include any offset information and is not tied to any specific time zone. It represents a date and time without time zone information.
- **ZonedDateTime:** ZonedDateTime includes both date, time, and time zone information. It represents an instant in time considering the time zone rules and daylight saving time changes.
- **Instant:** Instant represents a point in time in the UTC time zone, independent of any offset or time zone rules.

Java OffsetDateTime Class Declaration

Let's see the declaration of java.time.OffsetDateTime class

public **final class** OffsetDateTime **extends** Object

implements Temporal, TemporalAdjuster, Comparable<offsetdatetime>,

Serializable

Explanation

1. public final class OffsetDateTime extends Object

public final class: This denotes that OffsetDateTime is a public and final class in Java. Being a final class means that it cannot be subclassed or extended by other classes. This design decision ensures that the behavior of OffsetDateTime cannot be altered or overridden by subclasses.

OffsetDateTime: This is the name of the class. It represents a date-time with an offset from UTC and is part of the java.time package introduced in Java 8.

extends Object: In Java, all classes implicitly extend the Object class unless explicitly specified otherwise. The Object class is the root of the class hierarchy in Java, providing common methods like toString(), hashCode(), and equals() to all objects.

Java OffsetDateTime Class Declaration

Let's see the declaration of java.time.OffsetDateTime class

public **final class** OffsetDateTime **extends** Object

implements Temporal, TemporalAdjuster, Comparable<offsetdatetime>, Serializable

Explanation

2. implements Temporal, TemporalAdjuster, Comparable<OffsetDateTime>, Serializable

implements Temporal: This indicates that the OffsetDateTime class implements the Temporal interface. The Temporal interface is a part of the Date and Time API (java.time.temporal package) and represents a date-time object that can be manipulated.

implements TemporalAdjuster: OffsetDateTime also implements the TemporalAdjuster interface. This interface allows objects to be manipulated based on rules, such as adjusting to the next or previous day of the week.

Comparable<OffsetDateTime>: OffsetDateTime implements the Comparable interface with itself as the type parameter. This means instances of OffsetDateTime can be compared with each other using methods like compareTo(), enabling sorting and ordering operations.

Serializable: This interface indicates that instances of OffsetDateTime can be serialized, meaning they can be converted into a stream of bytes to be stored or transmitted and then deserialized back into objects. This allows OffsetDateTime instances to be persisted or sent across a network.

Methods of Java OffsetDateTime Class

int get(TemporalField field)	It is used to get the value of the specified field from this date-time as an int.
int getDayOfMonth()	It is used to get the day-of-month field.
iint getDayOfYear()	It is used to get the day-of-year field.
DayOfWeek getDayOfWeek()	It is used to get the day-of-week field, which is an enum DayOfWeek.
OffsetDateTime minusDays(long days)	It is used to return a copy of this OffsetDateTime with the specified number of days subtracted.
static OffsetDateTime now()	It is used to obtain the current date-time from the system clock in the default time-zone.
OffsetDateTime plusDays(long days)	It is used to return a copy of this OffsetDateTime with the specified number of days added.
LocalDate toLocalDate()	It is used to get the LocalDate part of this date-time.
Temporal adjustInto(Temporal temporal)	It adjusts the specified temporal object to have the same date and time as this object.
TZonedDateTime atZoneSameInstant(ZoneId zone)	It combines this date-time with a time-zone to create a ZonedDateTime ensuring that the result has the same instant.
TZonedDateTime atZoneSimilarLocal(ZoneId zone)	It combines this date-time with a time-zone to create a ZonedDateTime trying to keep the same local date and time.
Tint compareTo(OffsetDateTime other)	It compares this date-time to another date-time.
Tboolean equals(Object obj)	It checks if this date-time is equal to another date-time.
TString format(DateTimeFormatter formatter)	It formats this date-time using the specified formatter.
Tstatic OffsetDateTime from(TemporalAccessor temporal)	It obtains an instance of OffsetDateTime from a temporal object.
Tint getHour()	It gets the hour-of-day field.
Tlong getLong(TemporalField field)	It gets the value of the specified field from this date-time as a long.
TMonth getMinute()	It gets the minute-of-hour field.
TMonth getMonth()	It gets the month-of-year field using the Month enum.
Tint getMonthValue()	It gets the month-of-year field from 1 to 12.
Tint getNano()	It gets the nano-of-second field.

Java OffsetDateTime Class Example: getDayOfMonth()

```
//OffsetDateTimeExample1.java
import java.time.OffsetDateTime;
public class OffsetDateTimeExample1 {
    public static void main(String[] args) {
// Get the current OffsetDateTime
        OffsetDateTime offsetDT = OffsetDateTime.now();
// Print the day of the month
        System.out.println(offsetDT.getDayOfMonth());
    }
}
```

Output:

18

Java OffsetDateTime Class Example: getDayOfWeek()

OffsetDateTimeExample2.java

```
import java.time.OffsetDateTime;
public class OffsetDateTimeExample3 {
    // The main method is the entry point of the program.
    public static void main(String[] args) {
        // Get the current OffsetDateTime.
        OffsetDateTime offsetDT = OffsetDateTime.now();
        // Print the day of the week.
        System.out.println(offsetDT.getDayOfWeek());
    }
}
```

Output:

WEDNESDAY

java.time.OffsetTime Class in Java

Java OffsetTime class is an immutable date-time object that represents a time, often viewed as hour-minute-second offset.

OffsetTime class represents a time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 18:30:45+08:00, often viewed as an hour-minute-second-offset. This class is immutable and thread-safe, stores all time fields, to a precision of nanoseconds, as well as a zone offset.

Syntax: Class declaration

```
public final class OffsetTime extends Object implements Temporal,
 TemporalAdjuster, Comparable<OffsetTime>, Serializable
```

Now let us come to the main eccentricity to gathering knowledge of the methods of this class,
which is as depicted in tabular format below:

Method	Action Performed
format(DateTimeFormatter formatter)	Formats this time using the specified formatter.
of(LocalTime time, ZoneOffset offset)	Obtains an instance of OffsetTime from a local time and an offset.
range(TemporalField field)	Gets the range of valid values for the specified field.
toLocalTime()	Gets the LocalTime part of this date-time.
adjustInto(Temporal temporal)	Adjust the specified temporal object to have the same offset and time as this object.
atDate(LocalDate date)	Combine this time with a date to create an OffsetDateTime.
compareTo(OffsetTime other)	Compare this OffsetTime to another time.
equals(Object obj)	Check if this time is equal to another time.
format(DateTimeFormatter formatter)	Formats this time using the specified formatter.
from(TemporalAccessor temporal)	Obtains an instance of OffsetTime from a temporal object.
get(TemporalField field)	Gets the value of the specified field from this time as an int.
getHour()	Gets the hour-of-day field.
getLong(TemporalField field)	Gets the value of the specified field from this time as a long.
getMinute()	Gets the minute-of-hour field.
getNano()	Gets the nano-of-second field.
getOffset()	Gets the zone offset, such as ‘+01:00’.
getSecond()	Gets the second-in-minute field.

```
// Java Program to Implement OffsetTime class
// via now() method

// Importing required classes
import java.time.OffsetTime;
import java.time.temporal.ChronoField;

// Main class
public class GFG {

// Main driver method
public static void main(String[] args)
{

// Creating an object of class in main() method
OffsetTime offset = OffsetTime.now();
```

java.time.YearMonth Class in Java

Java **YearMonth class** provides the output of the format “year-month”. This class is an immutable class which means that it defines objects which, once created, never change their value. Any field which will be derived from a year and month, like quarter-of-year, are often obtained. This class does not store or represent a day, time or time-zone. For example, it would not show the value “November-2006-12:00” rather than it will show “2007-11”. It inherits the thing class and implements the Comparable interface.

Java YearMonth class implements Temporal, TemporalAdjuster, Comparable<YearMonth>, Serializable

public final class YearMonth extends Object implements Temporal, TemporalAdjuster, Comparable<YearMonth>, Serializable

Method	Description
adjustInto(Temporal temporal)	This method adjusts the specified temporal object to have this year-month.
atDay(int dayOfMonth)	This method combines this year-month with a day-of-month to create a LocalDate.
atEndOfMonth()	This method returns a LocalDate at the end of the month.
compareTo(YearMonth other)	This method compares this year-month to another year-month.
equals(Object obj)	This method checks if this year-month is equal to another year-month.
format(DateTimeFormatter formatter)	This method Formats this year-month using the specified formatter.
from(TemporalAccessor temporal)	This method obtains an instance of YearMonth from a temporal object.
get(TemporalField field)	This method gets the value of the specified field from this year-month as an int.
getLong(TemporalField field)	This method gets the value of the specified field from this year-month as along.

getMonth()	This method gets the month-of-year field using the Month enum.
getMonthValue()	This method gets the month-of-year field from 1 to 12.
getYear()	This method gets the year field.
hashCode()	A hash code for this year-month.
isAfter(YearMonth other)	This method checks if this year-month is after the specified year-month.
isBefore(YearMonth other)	This method checks if this year-month is before the specified year-month.
isLeapYear()	This method checks if the year is a leap year, according to the ISO proleptic calendar system rules.
isSupported(TemporalField field)	This method checks if the specified field is supported.
isSupported(TemporalUnit unit)	This method checks if the specified unit is supported.
isValidDay(int dayOfMonth)	This method checks if the day-of-month is valid for this year-month.
lengthOfMonth()	This method returns the length of the month, taking account of the year.
lengthOfYear()	This method returns the length of the year.
minus(long amountToSubtract, TemporalUnit unit)	This method returns a copy of this year-month with the specified amount subtracted.
minus(TemporalAmount	This method returns a copy of this year-month with the specified

Below is the implementation of some methods of the Java YearMonth class:

1. plus(): Returns a replica of this year-month with the required amount added.

```
// plus() Method implementation
import java.time.*;
public class Example {
    public static void main(String[] args)
    {
        YearMonth obj1 = YearMonth.now();

        // plus(Period.ofYears(int)) will add
        // no.of years to the existing year
        YearMonth obj2 = obj1.plus(Period.ofYears(0));
        System.out.println(obj2);
    }
}
```

Output 2021-02

2. minus(): Returns a replica of this year-month with the required amount subtracted.

```
// minus() method implementation
import java.time.*;
public class Example {
    public static void main(String[] args)
    {
        YearMonth obj1 = YearMonth.now();

        //.minus(Period.ofYears(int)) will subtract
        // no. ofyears from the existing year
        YearMonth obj2 = obj1.minus(Period.ofYears(3));
        System.out.println(obj2);
    }
}
```

java.time.Period Class in Java

The **Period** Class in Java class obtains a quantity or amount of time in terms of years, months and days. The time obtained is a date-based amount of time in the ISO-8601 calendar system, such as ‘4 years, 5 months, and 6 days. The units which are supported for a period are YEARS, MONTHS, and days. All these three fields are always present but may be set to zero.

Syntax: Class declaration

```
public final class Period extends Object implements ChronoPeriod,
Serializable
```

Below some the methods with the action performed by them are as follows in the tabular format are as follows:

addTo(Temporal temporal)	This method adds this period to the specified temporal object.
between(LocalDate startDateInclusive, LocalDate endDateExclusive)	This method obtains a Period consisting of the number of years, months, and days between two dates.
equals(Object obj)	This method checks if this period is equal to another period.
from(TemporalAmount amount)	This method obtains an instance of Period from a temporal amount.
get(TemporalUnit unit)	This method gets the value of the requested unit.
getChronology()	This method gets the chronology of this period, which is the ISO calendar system.
getDays()	This method gets the number of days of this period.
getMonths()	This method gets the number of months of this period.
getUnits()	This method gets the set of units supported by this period.
getYears()	This method gets the number of years of this period.
hashCode()	This method returns a hash code for this period.
isNegative()	This method checks if any of the three units of this period are negative.
isZero()	This method checks if all three units of this period are zero.
minus(TemporalAmount amountToSubtract)	This method returns a copy of this period with the specified period subtracted.
minusDays(long daysToSubtract)	This method returns a copy of this period with the specified days subtracted.

Implementation Example 1

```
// Java program to illustrate Period class
// demonstrate the methods of this class
// Methods - minus() and ofMonths()

// Importing Period class from
// java.time package
import java.time.Period;

// Main class
public class GFG {
// Main driver method
public static void main(String[] args)
      {
// Obtaining period representing number of months
// using of months() method by
// creating object of period class
Period p1 = Period.ofMonths(6);
```

Output P4M

Example 2

```
// Java program to illustrate Period class
// demonstrate the methods of this class
// Methods like ofDays() and addTo()

// Importing all classes from java.time package
import java.time.*;
import java.time.temporal.Temporal;

// Main class
public class GFG {

// Main driver method
public static void main(String[] args)
      {
// Getting a period representing number of days
// using ofDays() method
```

Method 1:
ofDays() method of this class is used to obtain a period from the given number of Days as a parameter.
Syntax:

```
public static Period ofDays(int numberOfDays)
```

Parameters: This method accepts a single parameter number of days which is the number of Days to be parsed into a Period object.
Returns: This function returns the period which is the Period object parsed with the given number of Days.

Method 2:
addTo() method of this class adds this Period to the specific temporal object.
Syntax:

```
public Temporal addTo(Temporal temporal)
```

Parameters: The temporal object to adjust should not be null.
Return Type: An object of the same type with the adjustment made.
Exceptions: DateTime and Arithmetic exceptions are thrown by this method.

Duration Class:

- Duration is a class in Java from the java.time package, introduced in Java 8.

- It represents a period of time between two points (such as between two LocalDateTime, LocalTime, or Instant), measured in nanoseconds, milliseconds, minutes, hours, etc.

Purpose of Duration:

- To handle temporal calculations involving a fixed period of time, such as calculating the difference between two dates/times or adding/ subtracting a specific duration from a date/time.

- It is immutable (once created, its value cannot be changed).

Core Methods of Duration:

- ofDays(): Creates a Duration representing a specific number of days.
- ofHours(): Creates a Duration representing a specific number of hours.
- ofMinutes(): Creates a Duration representing a specific number of minutes.
- ofSeconds(): Creates a Duration representing a specific number of seconds.
- ofMillis(): Creates a Duration representing a specific number of milliseconds.
- between(): Calculates the duration between two temporal objects (LocalDateTime, LocalTime, etc.).
- plus(): Adds a duration to a temporal object.
- minus(): Subtracts a duration from a temporal object.

Example

```
import java.time.LocalDateTime;
import java.time.Duration;

public class DurationExample {
    public static void main(String[] args) {
        // Creating a Duration of 5 hours
        Duration duration1 = Duration.ofHours(5);
        System.out.println("Duration in hours: " + duration1.toHours());  // Output: 5

        // Creating a Duration of 3 days and adding it to current date/time
        Duration duration2 = Duration.ofDays(3);
        LocalDateTime dateTime = LocalDateTime.now();
        LocalDateTime newDateTime = dateTime.plus(duration2);
        System.out.println("New Date and Time: " + newDateTime);

        // Calculating the duration between two dates
```

- Output:

- Duration in hours: 5
- New Date and Time: 2024-01-20T10:27:56.438203600
- Duration between two dates: 30 days

- Useful Features of Duration:

- Handles periods of time in nanoseconds, seconds, minutes, hours, days, etc.
- Supports arithmetic operations like adding or subtracting durations from temporal objects.
- Provides methods like toHours(), toMinutes(), toDays(), etc., for convenient retrieval of time measurements.