

JAVA Programación Orientada a Objetos

Java Time API

Java Time API es una poderosa API introducida en Java 8 para el manejo de fechas, horas y zonas horarias.

💡 El acrónimo API significa Interfaz de Programación de Aplicaciones (Application Programming Interface en inglés). Una API consiste en un conjunto de reglas y protocolos que facilitan la comunicación y la interacción entre distintos componentes de software. En el caso de la Java Time API, permite a los desarrolladores interactuar con el sistema y llevar a cabo operaciones relacionadas con fechas, tiempos y duraciones en sus aplicaciones

Clases principales

Una de las mejoras más significativas de la Java Time API es la clara distinción entre las diferentes formas de tiempo, cada una representada por su propia clase. Las principales clases que veremos son `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime` e `Instant`. Anteriormente, solo existía la clase `Date`, que carecía de métodos y clases de utilidad como las que tenemos ahora.

Utiliza esta información como referencia; no es necesario memorizar todos los métodos que se muestran aquí. Ten en cuenta que existen aún más métodos disponibles. Utiliza esta guía para explorar parte del potencial que esta API ofrece en la manipulación del tiempo a los programadores

LocalDate

LocalDate es una clase que representa una fecha sin tener en cuenta las horas ni los minutos. Resulta útil cuando solo se requiere la fecha, como en el caso de un cumpleaños o un aniversario.

Métodos estáticos:

- **of(int año, int mes, int día):** Crea una instancia de LocalDate con los valores de año, mes y día especificados.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
  
System.out.println(fecha); // Imprime: 2023-12-25
```

- **of(int año, Month mes, int día):** Similar al método anterior pero usando la enumeración Month para el mes.

```
LocalDate fecha = LocalDate.of(2023, Month.DECEMBER, 25);  
  
System.out.println(fecha); // Imprime: 2023-12-25
```

- **now():** Devuelve la fecha actual según la fecha del sistema.

```
LocalDate hoy = LocalDate.now();  
  
System.out.println(hoy); // Imprime la fecha actual
```

- **now(ZoneId zone):** Devuelve la fecha actual en una zona horaria específica.

```
LocalDate hoyEnTokio = LocalDate.now(ZoneId.of("Asia/Tokyo"));  
  
System.out.println(hoyEnTokio); // Imprime la fecha actual en  
Tokio
```

💡 **Un ZoneId** se emplea para identificar una región geográfica específica que sigue un conjunto de normas para el cambio de hora, lo cual incluye husos

horarios y horario de verano. Puede representar zonas horarias como "America/New_York", "Europe/Paris", "Asia/Tokyo", entre otras. Estas identificaciones se basan en el identificador de [zona horaria de la base de datos de la Autoridad de Números Asignados de Internet \(IANA\)](#).

- **parse(CharSequence text):** Crea una instancia de `LocalDate` a partir de una cadena de texto en formato yyyy-MM-dd.

```
LocalDate fecha = LocalDate.parse("2023-12-25");  
  
System.out.println(fecha); // Imprime: 2023-12-25
```

💡 En Java, un **CharSequence** es una interfaz que representa una secuencia de caracteres. Esta interfaz es implementada por clases como `String`, `StringBuilder` y `StringBuffer`, las cuales se utilizan para manipular y almacenar texto

Métodos de instancia:

- **isAfter(ChronoLocalDate other):** Comprueba si esta fecha es posterior a la especificada.

```
LocalDate fecha1 = LocalDate.of(2023, 12, 25);  
  
LocalDate fecha2 = LocalDate.of(2024, 1, 1);  
  
System.out.println(fecha2.isAfter(fecha1)); // Imprime: true
```

💡 **ChronoLocalDate** es una interfaz implementada por `LocalDate` y otras clases que representan fechas de distintos tipos de calendario, tales como `HijrahDate`, `JapaneseDate`, `MinguoDate` y `ThaiBuddhistDate`.

- **isBefore(ChronoLocalDate other):** Comprueba si esta fecha es anterior a la especificada.

```
LocalDate fecha1 = LocalDate.of(2023, 12, 25);  
LocalDate fecha2 = LocalDate.of(2024, 1, 1);  
System.out.println(fecha1.isBefore(fecha2)); // Imprime: true
```

- **isEqual(ChronoLocalDate other):** Comprueba si esta fecha es igual a la especificada.

```
LocalDate fecha1 = LocalDate.of(2023, 12, 25);  
LocalDate fecha2 = LocalDate.of(2023, 12, 25);  
System.out.println(fecha1.isEqual(fecha2)); // Imprime: true
```

- **isLeapYear():** Comprueba si el año es bisiesto.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
System.out.println(fecha.isLeapYear()); // Imprime: false
```

- **getMonth():** Obtiene el mes del año utilizando la enumeración Month.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
System.out.println(fecha.getMonth()); // Imprime: DECEMBER
```

- **getMonthValue():** Obtiene el valor del mes, de 1 a 12.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
System.out.println(fecha.getMonthValue()); // Imprime: 12
```

- **getYear():** Obtiene el año.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
  
System.out.println(fecha.getYear()); // Imprime: 2023
```

- **getDayOfMonth():** Obtiene el día del mes.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
  
System.out.println(fecha.getDayOfMonth()); // Imprime: 25
```

- **getDayOfWeek():** Obtiene el día de la semana utilizando la enumeración DayOfWeek.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
  
System.out.println(fecha.getDayOfWeek()); // Imprime: MONDAY si  
el 25 de diciembre de 2023 cae en lunes.
```

- **getDayOfYear():** Obtiene el día del año.

```
LocalDate fecha = LocalDate.of(2023, 12, 25);  
  
System.out.println(fecha.getDayOfYear()); // Imprime: 359 si el  
25 de diciembre es el día 359 del año. (por lo años bisiestos)
```

- **plusDays(long dias):** Devuelve una copia de esta fecha con la cantidad especificada de días añadidos.

```
LocalDate mañana = LocalDate.now().plusDays(1);  
  
System.out.println(mañana); // Imprime la fecha de mañana
```

- **plusMonths(long meses):** Devuelve una copia de esta fecha con la cantidad especificada de meses añadidos.

```
LocalDate enUnMes = LocalDate.now().plusMonths(1);
```

```
System.out.println(enUnMes); // Imprime la fecha en un mes
```

- **plusWeeks(long semanas):** Devuelve una copia de esta fecha con la cantidad especificada de semanas añadidas.

```
LocalDate enUnaSemana = LocalDate.now().plusWeeks(1);  
  
System.out.println(enUnaSemana); // Imprime la fecha en una  
semana
```

- **plusYears(long años):** Devuelve una copia de esta fecha con la cantidad especificada de años añadidos.

```
LocalDate enUnAño = LocalDate.now().plusYears(1);  
  
System.out.println(enUnAño); // Imprime la fecha en un año
```

- **minusDays(long dias):** Devuelve una copia de esta fecha con la cantidad especificada de días restados.

```
LocalDate ayer = LocalDate.now().minusDays(1);  
  
System.out.println(ayer); // Imprime la fecha de ayer
```

- **minusMonths(long meses):** Devuelve una copia de esta fecha con la cantidad especificada de meses restados.

```
LocalDate haceUnMes = LocalDate.now().minusMonths(1);  
  
System.out.println(haceUnMes); // Imprime la fecha de hace un  
mes
```

- **minusWeeks(long semanas):** Devuelve una copia de esta fecha con la cantidad especificada de semanas restadas.

```
LocalDate haceUnaSemana = LocalDate.now().minusWeeks(1);  
  
System.out.println(haceUnaSemana); // Imprime la fecha de hace
```

```
una semana
```

- **minusYears(long años):** Devuelve una copia de esta fecha con la cantidad especificada de años restados.

```
LocalDate haceUnAño = LocalDate.now().minusYears(1);  
  
System.out.println(haceUnAño); // Imprime la fecha de hace un  
año
```

- **atStartOfDay():** Combina esta fecha con la hora de medianoche para crear un LocalDateTime al comienzo de esta fecha.

```
LocalDateTime inicioDelDia = LocalDate.now().atStartOfDay();  
  
System.out.println(inicioDelDia); // Imprime la fecha y hora al  
comienzo del día
```

- **atStartOfDay(ZonedId zone):** Devuelve una fecha-hora con zona horaria de esta fecha a la hora más temprana válida según las reglas en la zona horaria.

```
ZonedDateTime inicioDelDiaEnTokio =  
LocalDate.now().atStartOfDay(ZoneId.of("Asia/Tokyo"));  
  
System.out.println(inicioDelDiaEnTokio); // Imprime la fecha y  
hora al inicio del día en Tokio
```

- **atTime(int hora, int minuto):** Combina esta fecha con una hora para crear un LocalDateTime.

```
LocalDateTime mediodia = LocalDate.now().atTime(12, 0);  
  
System.out.println(mediodia); // Imprime la fecha y la hora a  
las 12:00
```

- **atTime(int hora, int minuto, int segundo):** Similar al método anterior pero también se especifican los segundos.

- **withDayOfMonth(int diaDelMes):** Devuelve una copia de esta fecha con el día del mes alterado.

```
LocalDate primerDiaDelMes = LocalDate.now().withDayOfMonth(1);  
  
System.out.println(primerDiaDelMes); // Imprime el primer día  
del mes actual
```

- **withDayOfYear(int diaDelAño):** Devuelve una copia de esta fecha con el día del año alterado.

```
LocalDate primerDiaDelAño = LocalDate.now().withDayOfYear(1);  
  
System.out.println(primerDiaDelAño); // Imprime el primer día  
del año actual
```

- **withMonth(int mes):** Devuelve una copia de esta fecha con el mes del año alterado.

```
LocalDate enFebrero = LocalDate.now().withMonth(2);  
  
System.out.println(enFebrero); // Imprime la fecha establecida  
en febrero
```

- **withYear(int año):** Devuelve una copia de esta fecha con el año alterado.

```
LocalDate en2024 = LocalDate.now().withYear(2024);  
  
System.out.println(en2024); // Imprime la fecha establecida en  
el año 2024
```

- **until(ChronoLocalDate endDateExclusive):** Calcula el periodo entre esta fecha y otra fecha como un Period.

```
LocalDate hoy = LocalDate.now();  
  
LocalDate añoNuevo = LocalDate.of(hoy.getYear() + 1, 1, 1);
```



```
Period periodo = hoy.until(añoNuevo);

System.out.println(periodo); // Imprime el periodo hasta el año
nuevo
```

- **toString():** Devuelve esta fecha como una cadena de texto en formato yyyy-MM-dd.

```
LocalDate hoy = LocalDate.now();

System.out.println(hoy.toString()); // Imprime la fecha de hoy
en formato yyyy-MM-dd
```

LocalTime

LocalTime es una clase que representa una hora sin una fecha asociada. Es particularmente útil cuando solo se requiere la hora, como por ejemplo, la hora de inicio de un evento. Además, ten en cuenta que existen ejemplos de código que no se mostrarán aquí debido a que son similares a los vistos en LocalDate; la única diferencia radica en el tipo de dato utilizado.

Métodos estáticos:

- **of(int hour, int minute):** Obtiene una instancia de LocalTime a partir de una hora y minuto.

```
LocalTime mediodia = LocalTime.of(12, 0);

System.out.println(mediodia); // Imprime 12:00
```

- **of(int hour, int minute, int second):** Obtiene una instancia de LocalTime a partir de una hora, minuto y segundo.

```
LocalTime mediodiaConSegundos = LocalTime.of(12, 0, 30);

System.out.println(mediodiaConSegundos); // Imprime 12:00:30
```

- **of(int hour, int minute, int second, int nanoOfSecond):** Obtiene una instancia de `LocalTime` a partir de una hora, minuto, segundo y nanosegundo

```
LocalTime mediodiaConSegundosYNanos = LocalTime.of(12, 0, 30, 500);

System.out.println(mediodiaConSegundosYNanos);    // Imprime
12:00:30.000000500
```

- **now():** Obtiene la hora actual del reloj del sistema en la zona horaria predeterminada.

```
LocalTime ahora = LocalTime.now();
```

- **now(ZonedDateTime zone):** Obtiene la hora actual del reloj del sistema en la zona horaria especificada.
- **parse(CharSequence text):** Obtiene una instancia de `LocalTime` a partir de una cadena de texto en el formato estándar de 24 horas "HH:mm".

```
LocalTime hora = LocalTime.parse("10:15");

System.out.println(hora); // Imprime 10:15
```

Métodos de instancia:

- **isAfter(LocalTime other):** Comprueba si esta hora es posterior a la hora especificada.
- **isBefore(LocalTime other):** Comprueba si esta hora es anterior a la hora especificada.
- **isEqual(LocalTime other):** Comprueba si esta hora es igual a la especificada.
- **getHour():** Obtiene el campo de la hora del día.
- **getMinute():** Obtiene el campo del minuto de la hora.
- **getSecond():** Obtiene el campo del segundo del minuto.
- **getNano():** Obtiene el campo del nanosegundo de la hora.

- **plusHours(long hoursToAdd):** Devuelve una copia de este LocalTime con el número especificado de horas añadidas.
- **plusMinutes(long minutesToAdd):** Devuelve una copia de este LocalTime con el número especificado de minutos añadidos.
- **plusSeconds(long secondsToAdd):** Devuelve una copia de este LocalTime con el número especificado de segundos añadidos.
- **plusNanos(long nanosToAdd):** Devuelve una copia de este LocalTime con el número especificado de nanosegundos añadidos.

```
LocalTime horaAñadida = LocalTime.of(12, 0).plusNanos(500);
System.out.println(horaAñadida); // Imprime 12:00:00.000000500
```

- **minusHours(long hoursToSubtract):** Devuelve una copia de este LocalTime con el número especificado de horas restadas.
- **minusMinutes(long minutesToSubtract):** Devuelve una copia de este LocalTime con el número especificado de minutos restados.
- **minusSeconds(long secondsToSubtract):** Devuelve una copia de este LocalTime con el número especificado de segundos restados.
- **minusNanos(long nanosToSubtract):** Devuelve una copia de este LocalTime con el número especificado de nanosegundos restados.
- **atDate(LocalDate date):** Combina esta hora con una fecha para crear un LocalDateTime.

```
LocalDateTime fechaYHora =
LocalTime.now().atDate(LocalDate.of(2023, 7, 11));
System.out.println(fechaYHora); // Imprime la fecha y la hora
actual
```

- **atOffset(ZoneOffset offset):** Combina esta hora con un offset para crear un OffsetTime.

```
OffsetTime tiempoOffset =
LocalTime.now().atOffset(ZoneOffset.ofHours(-5));
```

```
System.out.println(tiempoOffset); // Imprime la hora actual con  
offset de -5 horas
```

💡 **ZoneOffset** es una clase que representa la diferencia de tiempo entre una zona horaria específica y el Tiempo Universal Coordinado (UTC), medida en horas y minutos. El Tiempo Universal Coordinado (UTC) es una escala de tiempo estándar utilizada como referencia para la medición del tiempo en todo el mundo. A diferencia de las zonas horarias basadas en ubicaciones geográficas específicas, UTC no está vinculado a ninguna región o país en particular. Por ejemplo, la zona horaria de Argentina se encuentra en UTC-3, mientras que en Colombia está en UTC-5. Por lo tanto, si en UTC son las 17:00, en Argentina serán las 14:00 y en Colombia las 12:00.

- **withHour(int hour):** Devuelve una copia de este LocalTime con la hora del día alterada.
- **withMinute(int minute):** Devuelve una copia de este LocalTime con el minuto de la hora alterado.
- **withSecond(int second):** Devuelve una copia de este LocalTime con el segundo del minuto alterado.
- **withNano(int nanoOfSecond):** Devuelve una copia de este LocalTime con el nanosegundo alterado.
- **toString():** Convierte esta hora a una cadena de texto, como 10:15.

LocalDateTime

LocalDateTime es una clase que combina tanto la fecha (LocalDate) como la hora (LocalTime). Resulta útil cuando se requieren tanto la fecha como la hora, como en el caso de la fecha y hora de inicio de un evento. Esta clase incorpora todos los métodos tanto de LocalDate como de LocalTime

Métodos instancia:

- **toLocalDate():** Devuelve la parte LocalDate de este LocalDateTime.

```
LocalDate fecha = LocalDateTime.now().toLocalDate();
```

- **toLocalTime():** Devuelve la parte LocalTime de este LocalDateTime.

```
LocalTime fecha = LocalDateTime.now().toLocalTime();
```

ZonedDateTime

ZonedDateTime es una clase que amplía la funcionalidad de LocalDateTime al proporcionar información de zona horaria en el formato ISO-8601, como por ejemplo '2018-12-09T20:30+01:00[Europe/Madrid]'. Esta clase combina la fecha y hora local con un desplazamiento de UTC y una identificación de la zona horaria. Es especialmente útil cuando se necesita representar la fecha y hora en una zona horaria específica

Métodos estáticos:

- **of(LocalDate date, LocalTime time, ZoneId zone):** Obtiene una instancia de ZonedDateTime a partir de una fecha local y una hora local.

```
ZonedDateTime zdt = ZonedDateTime.of(LocalDate.now(),  
LocalTime.now(), ZoneId.of("Europe/Paris"));  
  
System.out.println(zdt);
```

- **of(LocalDateTime localDateTime, ZoneId zone):** Obtiene una instancia de ZonedDateTime a partir de una fecha y hora local.

```
ZonedDateTime zdt = ZonedDateTime.of(LocalDateTime.now(),  
ZoneId.of("Europe/Paris"));  
  
System.out.println(zdt);
```

- **ofInstant(Instant instant, ZoneId zone):** Obtiene una instancia de ZonedDateTime a partir de un Instante.

```
ZonedDateTime zdt = ZonedDateTime.ofInstant(Instant.now(),
ZoneId.of("Europe/Paris"));

System.out.println(zdt);
```

- **ofInstant(LocalDateTime localDateTime, ZoneOffset offset, ZoneId zone):** Obtiene una instancia de ZonedDateTime a partir del instante formado al combinar la fecha-hora local y el desplazamiento.

```
ZonedDateTime zdt =
ZonedDateTime.ofInstant(LocalDateTime.now(), ZoneOffset.UTC,
ZoneId.of("Europe/Paris"));

System.out.println(zdt);
```

- **ofLocal(LocalDateTime localDateTime, ZoneId zone, ZoneOffset preferredOffset):** Obtiene una instancia de ZonedDateTime a partir de una fecha-hora local utilizando el desplazamiento preferido si es posible.

```
ZonedDateTime zdt = ZonedDateTime.ofLocal(LocalDateTime.now(),
ZoneId.of("Europe/Paris"), ZoneOffset.UTC);

System.out.println(zdt);
```

- **now():** Obtiene la fecha-hora actual del reloj del sistema en la zona horaria predeterminada.

```
ZonedDateTime zdt = ZonedDateTime.now();

System.out.println(zdt);
```

- **now(ZoneId zone):** Obtiene la fecha-hora actual del reloj del sistema en la zona horaria especificada.

```
ZonedDateTime zdt =
ZonedDateTime.now(ZoneId.of("Europe/Paris"));
```

```
System.out.println(zdt);
```

- **parse(CharSequence text):** Obtiene una instancia de ZonedDateTime a partir de una cadena de texto como "2007-12-03T10:15:30+01:00[Europe/Paris]".

```
ZonedDateTime zdt =  
ZonedDateTime.parse("2007-12-03T10:15:30+01:00[Europe/Paris]");  
System.out.println(zdt);
```

Métodos instancia:

- **toLocalDate():** Devuelve la parte LocalDate de este LocalDateTime.

```
LocalDate fecha = LocalDateTime.now().toLocalDate();
```

- **toLocalTime():** Devuelve la parte LocalTime de este LocalDateTime.

```
LocalTime fecha = LocalDateTime.now().toLocalTime();
```

- **toLocalDateTime():** Obtiene la parte de LocalDateTime de esta fecha-hora.

```
LocalDateTime ldt = ZonedDateTime.now().toLocalDateTime();  
System.out.println(ldt);
```

- **toOffsetDateTime():** Convierte esta fecha-hora a un OffsetDateTime.

```
OffsetDateTime odt = ZonedDateTime.now().toOffsetDateTime();  
System.out.println(odt);
```

💡 **OffsetDateTime** es similar a ZonedDateTime, pero no tiene una identificación de zona horaria, solo un desplazamiento desde UTC. Por ejemplo, "2023-07-11T12:00:00+02:00" es una OffsetDateTime

.toString(): Convierte esta fecha-hora a una cadena de texto, como 2007-12-03T10:15:30+01:00[Europe/Paris].

```
String fechaHora = ZonedDateTime.now().toString();  
System.out.println(fechaHora); // Imprime la fecha-hora actual
```

Instant

La clase Instant representa un instante específico en el tiempo. Los objetos Instant son útiles para la manipulación precisa de fechas y horas en aplicaciones que requieren una precisión de nanosegundos. Se considera la versión moderna de java.util.Date, pero con una mayor precisión. Es comúnmente utilizado para convertir objetos de la antigua clase Date a las nuevas clases de la API de Tiempo en Java. Al imprimir un objeto Instant, la salida se presenta en el formato ISO-8601, ofreciendo una representación clara de la fecha y hora.

```
2023-07-11T18:45:22.711Z
```

Esto indica que el instante específico es el 11 de julio de 2023, a las 18:45:22.711 en el horario UTC (la "Z" indica el horario UTC).

Por lo tanto, un Instant es esencialmente un valor numérico que representa un punto en el tiempo. Sin embargo, cuando se muestra, se formatea como una fecha y hora para facilitar su comprensión. Además, proporciona una representación numérica larga (long) que se obtiene mediante métodos como toEpochMilli() o getEpochSecond(), los cuales devuelven los segundos o milisegundos desde la 'época UNIX'.

💡 La **"época UNIX"** es el punto de inicio para medir el tiempo en sistemas UNIX y otros sistemas operativos y lenguajes de programación, incluyendo Java. **El punto de inicio es el instante 00:00:00 UTC del 1 de enero de 1970.** Es decir, cuando un sistema basado en UNIX representa una fecha y hora como un número, ese número se calcula como la cantidad de segundos que han transcurrido desde la medianoche UTC del 1 de enero de 1970. Por ejemplo, la época UNIX "946684800" representa el instante 00:00:00 UTC del 1 de enero de

2000. Y el número "1593485765" representa el instante 09:36:05 UTC del 30 de junio de 2020.

Métodos estáticos:

- **ofEpochMilli(long epochMilli):** Obtiene una instancia de Instant usando milisegundos desde la época de 1970-01-01T00:00:00Z.

```
long millisDesdeEpoch = 1626014282000L;

Instant instante = Instant.ofEpochMilli(millisDesdeEpoch);

System.out.println(instante);    // Imprime el instante
correspondiente a los milisegundos desde la época
```

- **ofEpochSecond(long epochSecond):** Obtiene una instancia de Instant usando segundos desde la época de 1970-01-01T00:00:00Z.

```
long segundosDesdeEpoch = 1626014282L;

Instant instante = Instant.ofEpochSecond(segundosDesdeEpoch);

System.out.println(instante);    // Imprime el instante
correspondiente a los segundos desde la época
```

- **ofEpochSecond(long epochSecond, long nanoAdjustment):** Obtiene una instancia de Instant usando segundos desde la época de 1970-01-01T00:00:00Z y la fracción de nanosegundos del segundo.

```
long segundosDesdeEpoch = 1626014282L;

long ajusteNanos = 500L;

Instant instante = Instant.ofEpochSecond(segundosDesdeEpoch,
ajusteNanos);

System.out.println(instante);    // Imprime el instante con
ajuste de nanosegundos
```

- **now():** Obtiene el instante actual desde el reloj del sistema.

Métodos de instancia:

- **isAfter(Instant otherInstant):** Verifica si este instante es posterior al instante especificado.
- **isBefore(Instant otherInstant):** Verifica si este instante es anterior al instante especificado.
- **getEpochSecond():** Obtiene el número de segundos desde la época de 1970-01-01T00:00:00Z.
- **getLong(TemporalField field):** Obtiene el valor del campo especificado de este instante como un long.

```
Instant instant = Instant.now();

System.out.println(instant.getLong(ChronoField.INSTANT_SECONDS)
); // imprimirá el número de segundos desde la época hasta el
instante actual.
```

💡 **TemporalField** es una interfaz en la API de Java Time que representa un campo de tiempo, como el año, mes, día, hora, minuto o segundo. Su implementación más utilizada es ChronoField.

- **getNano():** Obtiene el número de nanosegundos, más tarde en la línea de tiempo, desde el inicio del segundo.
- **plusSeconds(long secondsToAdd):** Devuelve una copia de este instante con la duración especificada en segundos añadida.
- **plusMillis(long millisToAdd):** Devuelve una copia de este instante con la duración especificada en milisegundos añadida.
- **plusNanos(long nanosToAdd):** Devuelve una copia de este instante con la duración especificada en nanosegundos añadida.
- **minusSeconds(long secondsToSubtract):** Devuelve una copia de este instante con la duración especificada en segundos restada.

- **minusMillis(long millisToSubtract):** Devuelve una copia de este instante con la duración especificada en milisegundos restada.
- **minusNanos(long nanosToSubtract):** Devuelve una copia de este instante con la duración especificada en nanosegundos restada.
- **toEpochMilli():** Convierte este instante al número de milisegundos desde la época de 1970-01-01T00:00:00Z.
- **toString():** Una representación de cadena de este instante usando la representación ISO-8601.

Convertir de Date a clases de la Java Time API y viceversa

Para convertir la clase `Date` a las clases de la Java Time API, utilizaremos `Instant` como intermediario, ya que `Date` tiene un método que permite crear una instancia de esta clase. `Instant` representa un punto en el tiempo y se puede considerar como un valor absoluto de tiempo.

```
Date date = new Date();
// Convertir de Date a Java Time API
Instant instant = date.toInstant();
LocalDate localDate = LocalDate.ofInstant(date.toInstant(),
ZoneId.systemDefault());
LocalTime localTime = LocalTime.ofInstant(date.toInstant(),
ZoneId.systemDefault());
LocalDateTime localDateTime = LocalDateTime.ofInstant(instant,
ZoneId.systemDefault());
ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(instant,
ZoneId.systemDefault());

// Convertir de Java Time API a Date
instant =
localDate.atTime(localTime).atZone(ZoneId.systemDefault()).toInstant();
instant =
localTime.atDate(localDate).atZone(ZoneId.systemDefault()).toInstant();
instant = localDateTime.atZone(ZoneId.systemDefault()).toInstant();
instant = zonedDateTime.toInstant();

date = Date.from(instant);
```

Unidades, duraciones y periodos de tiempo

Las clases `Period`, `Duration` y `ChronoUnit` en Java se utilizan para modelar cantidades y unidades de tiempo. `Period` es óptimo para trabajar con fechas en

términos de años, meses y días. Por otro lado, Duration es más adecuado para trabajar con tiempos exactos en horas, minutos, segundos y nanosegundos. ChronoUnit, en cambio, define una estándar para las unidades de tiempo, permitiendo expresar dichas unidades en diferentes contextos de manera consistente, desde nanosegundos hasta milenios, facilitando así el manejo del tiempo en diversas aplicaciones.

Period

En Java, un objeto Period representa una cantidad de tiempo en términos de años, meses y días.

Métodos estáticos:

- **between(LocalDate startDateInclusive, LocalDate endDateExclusive):** Obtiene un objeto Period que representa el número de años, meses y días entre dos fechas.
- **of(int years, int months, int days):** Crea un objeto Period que representa una cantidad específica de años, meses y días.
- **ofDays(int days):** Crea un objeto Period que representa un número de días.
- **ofMonths(int months):** Crea un objeto Period que representa un número de meses.
- **ofWeeks(int weeks):** Crea un objeto Period que representa un número de semanas. Internamente, las semanas se convierten en días.
- **ofYears(int years):** Crea un objeto Period que representa un número de años.
- **parse(CharSequence text):** Crea un objeto Period a partir de una cadena de texto con formato ISO-8601 para periodos (PnYnMnD).

```
Period periodo = Period.parse("P2Y6M15D");// 2 años 6 meses 15 días  
  
System.out.println(periodo); // Imprime: P2Y6M15D
```

Métodos de instancia:

- **Temporal addTo(Temporal temporal):** Este método añade este periodo al objeto temporal especificado. Por ejemplo:

```
LocalDate fecha = LocalDate.of(2020, 1, 1);  
  
Period periodo = Period.ofYears(1);  
  
fecha = (LocalDate) periodo.addTo(fecha);  
  
System.out.println(fecha); // Imprime: 2021-01-01
```

💡 **Temporal** es una interfaz base para las clases de fecha, hora y offset dentro de la API de fecha y hora de Java. Sus implementaciones son `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetTime`, `OffsetDateTime`, `Year`, `YearMonth`, `MonthDay`, `Instant`, `Duration` y `Period`.

- **boolean isNegative():** Este método verifica si alguna de las tres unidades de este periodo son negativas.
- **boolean isZero():** Este método verifica si las tres unidades de este periodo son cero.
- **Period multipliedBy(int scalar):** Este método devuelve una nueva instancia con cada elemento de este periodo multiplicado por el escalar especificado.

```
Period periodo = Period.of(1, 2, 3);  
  
periodo = periodo.multipliedBy(2);  
  
System.out.println(periodo); // Imprime: P2Y4M6D
```

- **Period negated():** Este método devuelve una nueva instancia con cada cantidad en este periodo negada.

```
Period periodo = Period.of(1, 2, 3);
```

```
periodo = periodo.negated();  
  
System.out.println(periodo); // Imprime: P-1Y-2M-3D
```

- **Period normalized():** Este método devuelve una copia de este periodo con los años y meses normalizados.

```
Period periodo = Period.of(1, 13, 3);  
  
periodo = periodo.normalized();  
  
System.out.println(periodo); // Imprime: P2Y1M3D
```

💡 **Un Period** normalizado tiene sus valores ajustados para que sean consistentes y no haya solapamiento innecesario. Por ejemplo, si tienes un periodo de 1 año y 13 meses, esto es esencialmente equivalente a 2 años y 1 mes, ya que hay 12 meses en un año. Por lo tanto, el método `normalized()` ajustará el periodo a 2 años y 1 mes.

- **long toTotalMonths():** Este método obtiene el total de meses en este periodo.

```
Period periodo = Period.of(2, 3, 3);  
  
System.out.println(periodo.toTotalMonths()); // Imprime: 27
```

- **Métodos `get()`, `minus()`, `plus()` y `with()`** correspondientes para las propiedades de años, meses y días.

Duration

En Java, un objeto `Duration` representa una cantidad de tiempo en términos de segundos y nanosegundos.

Métodos estáticos:

- **`between(Temporal startInclusive, Temporal endExclusive)`:** Este método crea un objeto `Duration` que representa la duración entre dos objetos temporales.

```
Instant inicio = Instant.now();

Thread.sleep(1000); // Esperamos 1 segundo

Instant fin = Instant.now();


Duration duracion = Duration.between(inicio, fin);

System.out.println(duracion.getSeconds()); // Imprimirá 1
```

- **`of(long amount, TemporalUnit unit)`:** Crea un objeto `Duration` que representa una cantidad en la unidad especificada.

```
Duration duracion = Duration.of(5, ChronoUnit.HOURS);

System.out.println(duracion.toMinutes()); // Imprimirá 300
```

 **TemporalUnit** es una interfaz en la API de fecha y hora de Java que representa una unidad de tiempo. Esta interfaz define cómo se calcula una cantidad de tiempo en términos de esta unidad. La implementación de esta interfaz es la clase `ChronoUnit`, la cual exploraremos en la siguiente sección.

- **`ofDays(long days)`:** Crea un objeto `Duration` que representa un número de días estándar de 24 horas.
- **`ofHours(long hours)`:** Crea un objeto `Duration` que representa un número de horas estándar.

- **ofMillis(long millis):** Crea un objeto Duration que representa un número de milisegundos.
- **ofMinutes(long minutes):** Obtiene una Duration que representa un número de minutos estándar.
- **ofSeconds(long seconds):** Obtiene una Duration que representa un número de segundos
- **ofSeconds(long seconds, long nanoAdjustment):** Crea un objeto Duration que representa un número de segundos y un ajuste en nanosegundos.
- **parse(CharSequence text):** Crea un objeto Duration de una cadena de texto como PnDTnHnMn.nS.

```
Duration duracion = Duration.parse("PT20.345S");  
System.out.println(duracion.toMillis()); // Imprimirá 20345
```

Métodos de instancia:

- **abs():** Devuelve una copia de esta duración con una longitud positiva.
- **addTo(Temporal temporal):** Agrega esta duración al objeto temporal especificado.
- **dividedBy(long divisor):** Devuelve una copia de esta duración dividida por el valor especificado.
- **multipliedBy(long multiplicand):** Devuelve una copia de esta duración multiplicada por el escalar.
- **isNegative():** Verifica si esta duración es negativa, excluyendo cero.
- **isZero():** Verifica si esta duración es de longitud cero.
- **negated():** Devuelve una copia de esta duración con la longitud negada.
- **toDays():** Obtiene el número de días en esta duración.
- **toHours():** Obtiene el número de horas en esta duración.
- **toMillis():** Convierte esta duración a la longitud total en milisegundos.
- **toMinutes():** Obtiene el número de minutos en esta duración.

- **toNanos():** Convierte esta duración a la longitud total en nanosegundos expresada como un long.
- **Métodos get() with()** con las propiedades de segundos y nanosegundos.
- **Métodos minus(), plus()** correspondientes para las propiedades de días, horas, minutos, segundos, milisegundos y nanosegundos..

ChronoUnit

En Java, ChronoUnit es un Enum con propiedades y métodos que representan una unidad estándar de tiempo, que proporciona una amplia gama de opciones para expresar medidas de tiempo más allá de las horas, minutos y segundos, incluyendo también días, semanas, meses, años, décadas, siglos y milenios.

Constantes:

- **DAYS:** Unidad que representa el concepto de un día.
- **DECADES:** Unidad que representa el concepto de una década.
- **ERAS:** Unidad que representa el concepto de una era.
- **FOREVER:** Unidad artificial que representa el concepto de eternidad.
- **HALF_DAYS:** Unidad que representa el concepto de medio día, como se usa en AM/PM.
- **HOURS:** Unidad que representa el concepto de una hora.
- **MICROS:** Unidad que representa el concepto de un microsegundo.
- **MILLENNIA:** Unidad que representa el concepto de un milenio.
- **MILLIS:** Unidad que representa el concepto de un milisegundo.
- **MINUTES:** Unidad que representa el concepto de un minuto.
- **MONTHS:** Unidad que representa el concepto de un mes.
- **NANOS:** Unidad que representa el concepto de un nanosegundo, la unidad de tiempo más pequeña soportada.
- **SECONDS:** Unidad que representa el concepto de un segundo.
- **WEEKS:** Unidad que representa el concepto de una semana.

- **YEARS:** Unidad que representa el concepto de un año.

Métodos de las Constantes:

- **addTo(R temporal, long amount):** Devuelve una copia del objeto temporal especificado con el período especificado agregado.

```
LocalTime localTime = LocalTime.of(10,15,30);  
  
Temporal temporal = ChronoUnit.HOURS.addTo(localTime, 2);  
  
System.out.println(temporal); // 12:15:30
```

- **between(Temporal temporal1Inclusive, Temporal temporal2Exclusive):** Calcula la cantidad de tiempo entre dos objetos temporales.

```
LocalTime start = LocalTime.of(10, 15, 30);  
  
LocalTime end = LocalTime.of(13, 20, 40);  
  
long hours = ChronoUnit.HOURS.between(start, end);  
  
System.out.println(hours); // 3
```

- **getDuration():** Obtiene la duración estimada de esta unidad en el sistema de calendario ISO.

```
Duration duration = ChronoUnit.HOURS.getDuration();  
  
System.out.println(duration); // PT1H
```

- **isDateBased():** Comprueba si esta unidad es una unidad de fecha.

```
boolean isDateBased = ChronoUnit.DAYS.isDateBased();  
  
System.out.println(isDateBased); // true
```

- **isDurationEstimated():** Comprueba si la duración de la unidad es una estimación.

```
boolean isDurationEstimated = ChronoUnit.HOURS.isDurationEstimated();  
  
System.out.println(isDurationEstimated); // false
```

- **isSupportedBy(Temporal temporal):** Comprueba si esta unidad es compatible con el objeto temporal especificado.

```
LocalDateTime localDateTime = LocalDateTime.now();  
  
boolean isSupported = ChronoUnit.YEARS.isSupportedBy(localDateTime);  
  
System.out.println(isSupported); // true
```


- **isTimeBased():** Comprueba si esta unidad es una unidad de tiempo del día.

```
boolean isTimeBased = ChronoUnit.YEARS.isTimeBased();  
  
System.out.println(isTimeBased); // false
```

- **toString():** Devuelve el nombre de esta constante enum, tal como está contenido en la declaración.

Formateo y Parsing

Las clases de la API de Tiempo de Java **pueden ser formateadas en cadenas y analizadas desde cadenas**. Este proceso es útil para mostrar fechas y horas a los usuarios, así como para leer fechas y horas de los datos de entrada del usuario. Para realizar estas operaciones, **se utiliza la clase DateTimeFormatter**, la cual proporciona métodos para formatear y analizar fechas y horas de acuerdo con patrones personalizados o estilos predefinidos

Te invitamos a ver el siguiente video que te proporcionará una mejor comprensión sobre cómo utilizar clase DateTimeFormatter:  [Time Api | DateTimeFormatter | JAVA | Egg](#)

Formateo

El formateo se utiliza para **convertir un objeto de fecha y hora en una cadena**, generalmente para mostrar al usuario. Por ejemplo, supongamos que tienes un objeto `LocalDate` que deseas mostrar en el formato "dd/MM/yyyy".

```
LocalDate fecha = LocalDate.of(2023, 1, 23);
DateTimeFormatter formateador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String fechaFormateada = fecha.format(formateador); // Formateo
System.out.println(fechaFormateada); // Imprime 23/01/2023
```

En este ejemplo, `DateTimeFormatter.ofPattern("dd/MM/yyyy")` crea un formateador con el patrón especificado. Luego, `fecha.format(formateador)` convierte el objeto `LocalDate` en una cadena según ese patrón.

Parsing

El parsing es el **proceso opuesto al formateo**. Se utiliza para **convertir una cadena en un objeto de fecha y hora**, generalmente para leer la entrada del usuario o datos de un archivo. Por ejemplo, supongamos que tienes una cadena "23/01/2023" que deseas convertir en un objeto `LocalDate`.

```
String cadenaFecha = "23/01/2023";
DateTimeFormatter formateador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate fecha = LocalDate.parse(cadenaFecha, formateador); // Parsing
System.out.println(fecha); // Imprime 2023-01-23
```

En este ejemplo, `LocalDate.parse(cadenaFecha, formateador)` convierte la cadena en un objeto `LocalDate` según el patrón del formateador.

Algunos patrones comunes que podrías utilizar con `DateTimeFormatter.ofPattern()` incluyen:

- **"dd/MM/yyyy"**: Día de dos dígitos, mes de dos dígitos y año de cuatro dígitos.
- **"MM/dd/yyyy"**: Mes de dos dígitos, día de dos dígitos y año de cuatro dígitos.
- **"yyyy-MM-dd HH:mm"**: Año de cuatro dígitos, mes de dos dígitos, día de dos dígitos, hora de dos dígitos y minuto de dos dígitos.

Es importante recordar que el parsing es sensible al formato. Si la cadena no coincide con el patrón especificado por el formateador, se lanzará una excepción `DateTimeParseException`. Por lo tanto, siempre asegúrate de que la cadena que estás parseando tenga el formato correcto.

Aquí te dejamos el enlace a la documentación de Java donde puedes aprender cómo construir patrones de fecha: [documentación DateTimeFormatter](#)