

Trabajar con fechas y horas en Java

Fuente: <https://www.campusmvp.es/recursos/post/como-manejar-correctamente-fechas-en-java-el-paquete-java-time.aspx>

El paquete `java.time`

Las **clases oficiales para manejo de fechas en Java** se encuentran incluidas en el paquete `java.time`.

Las clases de este paquete solucionan la mayor parte de los problemas de las clases tradicionales, e incluyen soporte automático para cosas como **años bisiestos, zonas horarias y cambio automático de zona horaria**.

Este paquete `java.time` incluye muchas clases, pero las básicas son:

- **LocalDate**: representa a fechas sin la hora y nos facilita su manejo para declararlas, sumar y restar fechas y compararlas.
- **LocalTime**: es idéntica a la anterior, pero para el manejo de horas, sin ninguna fecha asociada, pudiendo así compararlas, sumar o restar tiempo a las mismas.
- **LocalDateTime**: es una combinación de las dos anteriores, que permite hacer lo mismo con fechas y horas simultáneamente.
- **Instant**: es muy parecida a la anterior, pero a la vez muy diferente. Se usa para almacenar un punto determinado en el tiempo, o sea con fecha y hora, pero guarda su valor como un *timestamp* de UNIX, es decir, en nanosegundos desde el *epoch* de UNIX (1/1/1970 a las 00:00) y usando la zona horaria UTC. Es muy útil para manejar momentos en el tiempo de manera neutra e intercambiarlo entre aplicaciones y sistemas, por lo que lo verás utilizado muy a menudo.
- **ZonedDateTime**: esta clase es como la `LocalDateTime` pero teniendo en cuenta una zona horaria concreta, ya que las anteriores no la tienen en cuenta.

- **Period**: esta clase auxiliar nos ayuda a obtener diferencias entre fechas en distintos periodos (segundos, minutos, días...) y también a añadir esas diferencias a las fechas.
- **Duration**: esta es muy parecida a la anterior, pero para manejo de horas exclusivamente.

Construyendo fechas y horas con java.time

Estas **clases son estáticas** y dado que carecen de constructores públicos, se instancian usando métodos de tipo "factoría o estáticos", es decir, tienen métodos que construyen estas clases a partir de posibles parámetros que le pasemos.

En concreto, todas las de manejo de fechas y horas disponen de tres métodos importantes, que son:

- **now()**: crean instancias nuevas a partir de la fecha y hora actual.
- **of()**: construyen fechas y horas a partir de sus partes.
- **with()**: modifican la fecha u hora actual en función del parámetro que se le pase, con alguna cantidad (años, días, horas...) o alguna clase de ajuste que enseguida estudiaremos.

Vamos a ver `now()` en acción con algunas de estas clases:

```
System.out.println("La fecha actual es: " + LocalDate.now());
System.out.println("La hora actual es: " + LocalTime.now());
System.out.println("La fecha y hora actuales son: " + LocalDateTime.now());
System.out.println("El instante actual es: " + Instant.now());
System.out.println("La fecha y hora actuales con zona horaria son: " +
ZonedDateTime.now());
```

Que nos mostrarían por pantalla algo similar a esto:

```
La fecha actual es: 2020-07-06
La hora actual es: 18:36:53.808065
La fecha y hora actuales son: 2020-07-06T18:36:53.809963
El instante actual es: 2020-07-06T18:36:53.810937Z
La fecha y hora actuales con zona horaria son: 2020-07-06T18:36:53.895234Z[Etc/UTC]
```

Fíjate en cómo cada tipo de clase genera un tipo de dato ligeramente diferente.

Al convertirlas a cadena para mostrarlas se generan en el formato ISO 8601, que es un estándar ampliamente aceptado. Luego veremos cómo formatearlas de otro modo que nos interese más.

Para controlar qué fechas y horas generamos podemos usar el método estático `of()` que admite ciertos parámetros en función del tipo de dato utilizado. Por ejemplo:

```
System.out.println("Fecha de mi cumpleaños: "+LocalDate.of(1972,Month.MAY,23));
```

```
Fecha de mi cumpleaños: 1972-05-23
```

Fíjate en que para el mes, aunque podría haber utilizado los números del 1 al 12 para indicarlo, he usado una enumeración específica que existe para ello llamada `Month`, cuyos miembros son los nombres de los meses en inglés. Así que mayo, que sería el mes 5, se convierte en `Month.MAY`.

En el caso de `LocalDateTime` también le podríamos haber pasado la hora llegando hasta los nanosegundos, en sucesivos parámetros:

```
System.out.println("Con la hora exacta: "+LocalDateTime.of(1972,Month.MAY,23,20,01,15,0023));
```

```
Con la hora exacta: 1972-05-23T20:01:15.000000019
```

que incluiría la hora exacta. Este método [tiene varias sobrecargas](#), por lo que, por ejemplo, le podríamos haber pasado también dos parámetros, uno con la fecha y otro con la hora usando sendas clases `LocalDate` y `LocalTime`.

Si intentas pasar una fecha incorrecta al método estático se produce una excepción. Por ejemplo, si quisieras usar el día 29 de febrero de 2019:

```
System.out.println("Día bisiesto de 2019: " + LocalDate.of(2019,Month.FEBRUARY,29));
```

Dado que ese año no fue bisiesto, se produciría una excepción:

```
Exception in thread "main" java.time.DateTimeException: Invalid date 'February 29' as '2019'
is not a leap year
    at java.base/java.time.LocalDate.create(LocalDate.java:457)
    at java.base/java.time.LocalDate.of(LocalDate.java:251)
    at Main.main(Main.java:24)
exit status 1
```

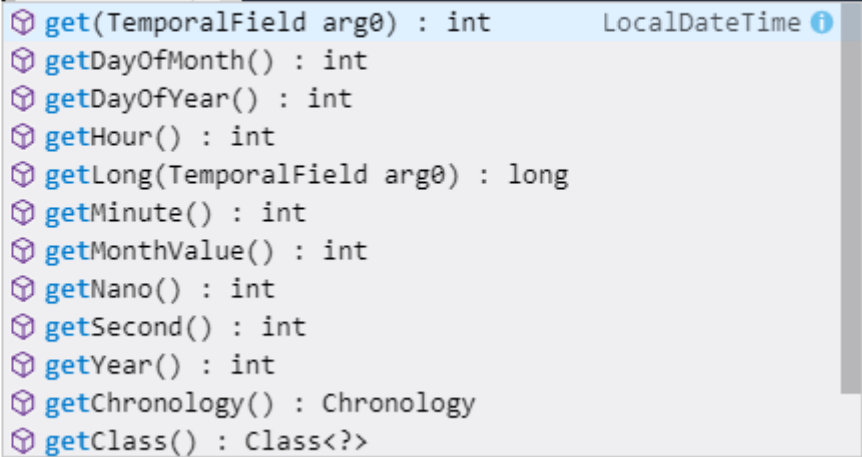
Partes de una fecha o una hora

Es posible extraer cualquier parte de una fecha o una hora a través de los métodos `getXXX()` que ofrecen estas clases.

Por ejemplo, `getHour()`, `getMinute()`, `getMonth()` o `getDayOfWeek()`.

```
5T20:40:15");
" + hoyConHora.get();

Formatter.ofPattern
+ seisNov);
```



Gracias a ellos podemos extraer cualquier parte de una fecha para trabajar con ella.

Transformando fechas y horas

O lo que es lo mismo: construyendo unas fechas y horas a partir de otras. Según la clase que manejemos tendremos una serie de métodos para añadir o quitar intervalos al dato.

Por ejemplo, la clase `LocalDate` dispone de los métodos:

- `plusDays()` / `minusDays()`: para sumar o restar días a la fecha.
- `plusWeeks()` / `minusWeeks()`: ídem con semanas.
- `plusMonths()` / `minusMonths()`: para añadir o quitar meses.
- `plusYears()` / `minusYears()`: para sumar o restar años.

Del mismo modo `LocalTime` ofrece los métodos `plusNanos()`, `plusSeconds()`, `plusMinutes()` y `plusHours()` para sumar nanosegundos, segundos, minutos y horas a la hora actual respectivamente. Del mismo modo, tiene los mismos métodos, pero con el prefijo `minus`, para restarlas: `minusNanos()`, etc. Intuitivo y directo:

```
System.out.println("La fecha dentro de 10 días: " +  
    LocalDate.now().plusDays(10));  
System.out.println("La fecha y hora de hace 32 horas: " +  
    LocalDateTime.now().minusHours(32));
```

```
Suma y resta de periodos de tiempo:  
La fecha dentro de 10 días: 2020-07-18  
La fecha y hora de hace 32 horas: 2020-07-07T00:08:39.748610
```

Dado que son clases estáticas, recuerda, lo que se devuelve en todos los casos son instancias nuevas del nuevo dato, no versiones modificadas de los datos originales.

Truco: en realidad, si a `plusXXX()` le pasas un número negativo estarás consiguiendo el mismo efecto que si usas el método correspondiente `minusXXX()`.

Ajustadores temporales

Aparte de los métodos de suma y resta de fechas que acabamos de ver, existe una clase especializada llamada `TemporalAdjuster` que nos permite definir ajustes para las fechas para obtener nuevas fechas a partir de una existente.

Existe una clase estática llamada `TemporalAdjusters` (en plural) [cuyos métodos](#) permiten obtener ajustes de fecha (de la clase `TemporalAdjuster`) de manera sencilla para hacer muchas cosas.

Por ejemplo, si queremos obtener el primer día del mes de una determinada fecha podemos usar el método `TemporalAdjusters.firstDayOfMonth()`. Del mismo modo existen "ajustadores" para otras operaciones similares, que puedes ver en el enlace anterior.

Así, para averiguar la fecha del primer día del mes que viene, podemos escribir:

```
System.out.println("El primer día del próximo mes es: " +  
    LocalDate.now().with(  
        TemporalAdjusters.firstDayOfNextMonth()  
    ).getDayOfWeek());
```

```
El primer día del próximo mes es: SATURDAY
```

Lo que hace es utilizar **el método** `with()` de las clases de tiempo (del que hablamos al principio) que en este caso toma un ajustador para obtener el primer día del mes siguiente. Luego usamos el método `getDayOfWeek()` sobre la fecha resultante para saber qué día de la semana es.

O para saber la fecha del último día del mes actual:

```
System.out.println("El último día de este mes es: " +  
    LocalDate.now().with(  
        TemporalAdjusters.lastDayOfMonth())
```

```
El último día de este mes es: 2020-07-31
```

Por supuesto, como lo que devuelven son objetos de tiempo, se pueden combinar con las funciones `plusXXX()` y `minusXXX()` para hacer más operaciones.

Es importante señalar que, se pueden crear nuestros propios ajustadores temporales con tan solo implementar una interfaz. Puedes ver un ejemplo [en la propia documentación de Java](#).

Tiempo transcurrido entre fechas y horas

Para comparar horas podemos hacerlo con los métodos `isAfter` e `isBefore`.

boolean	<code>isAfter(LocalTime other)</code> Comprueba si esta hora es posterior a la hora especificada.
boolean	<code>isBefore(LocalTime other)</code> Comprueba si esta hora es anterior a la hora especificada.

Otra tarea habitual que necesitaremos hacer es obtener **la diferencia entre dos fechas u horas**, o sea, el tiempo transcurrido entre dos instantes de tiempo.

Para ello existe una interfaz `java.time.temporal.TemporalUnit`, una enumeración `ChronoUnit` y una clase `Period` en ese mismo paquete que se encargan de facilitarnos la vida para esto. Con sus métodos: `between()` y `until()` nos proporcionan respectivamente el tiempo transcurrido entre dos instantes de tiempo y el tiempo que falta para llegar a una fecha u hora determinadas. Vamos a verlo.

Por ejemplo, imaginemos que queremos saber cuánto tiempo ha transcurrido entre la fecha de tu nacimiento y el día de hoy. Para averiguarlo sólo hay que hacer algo como esto:

```
LocalDate fNacimiento = LocalDate.of(1972, Month.MAY, 23);
System.out.println("Mi edad es de " +
ChronoUnit.YEARS.between(fNacimiento, LocalDate.now())
+ " años."
);
```

```
Mi edad es de 48 años.
```

La clase `ChronoUnit` dispone de [una serie de constantes](#) que nos permiten obtener las unidades que nos interesen (que a su vez son también objetos de la clase `ChronoUnit`) y que, con su método `between()` nos permiten obtener el intervalo que nos interese. En este caso un número que representa la cantidad de años entre la fecha de mi nacimiento y el día de hoy, o sea, mi edad.

Si quisiésemos, por ejemplo, saber cuánto tiempo falta para llegar a final de año, podemos sacar partido a la clase `Period` para lograrlo:

```
LocalDate hoy = LocalDate.now();
LocalDate finDeAño = hoy.with(TemporalAdjusters.lastDayOfYear());
Period hastaFinDeAño = hoy.until(finDeAño);
int meses = hastaFinDeAño.getMonths();
int dias = hastaFinDeAño.getDays();
System.out.println("Faltan " + meses + " meses y " + dias + "
días hasta final de año."
);
```

```
Faltan 5 meses y 23 días hasta final de año.
```

La clase `Period` también dispone del método estático `between()` para obtener el periodo entre dos elementos de tiempo, por lo que la línea 3 anterior se podría sustituir por esta:

```
Period hastaFinDeAño = Period.between(hoy, finDeAño);
```

y todo funcionaría igual.

Puedes conocer otros métodos de `Period` en la [documentación oficial de Java](#). Verás que son sencillos de utilizar.

"Parseando" fechas

Una tarea habitual en el manejo de fechas es "parsearlas", es decir, **interpretarlas a partir de una cadena**, generalmente recibida de la entrada de un usuario o de algún sistema de almacenamiento externo.

Las clases de `java.time`, por fortuna, ofrecen el método `parse()` que se ocupa de esto de manera trivial. Tiene dos sobrecargas, una que recibe la cadena a interpretar y, una segunda que además añade un formateador especializado si lo necesitamos:

```
LocalDate hoy = LocalDate.parse("2020-07-06");  
LocalDate seisNov = LocalDate.parse("6/11/2020",  
DateTimeFormatter.ofPattern("d/M/yyyy"));
```

En el primer caso se interpreta el formato por defecto, ISO 8601, **el cual podría incluir la hora** si usamos una clase que contemple horas también.

En el segundo caso usamos un formato propio mediante la clase `DateTimeFormatter` y su método `ofPattern()`, que es un método estático para obtener un formateador a partir de una cadena basada [en letras estándar para formato](#).

Formato personalizado de fechas

Del mismo modo que podemos usar la clase `DateTimeFormatter` para interpretar cadenas de texto y convertirlas a fechas, también le sacaremos partido para la operación contraria: **convertir una clase temporal en una cadena de texto usando el formato que nos interese**.

Para ello sólo tenemos que usar **el método `format()`** que todas ellas poseen y pasarle un formateador.

Este código muestra el aspecto de una fecha con su conversión a texto por defecto (ISO 8601), en ese mismo formato pero indicándolo explícitamente y en formato español (día del mes antes del mes y los elementos separados por barras) indicando ese formato manualmente:

```
System.out.println("Formato por defecto: " + fechaConHora);
System.out.println("Formato ISO 8601 (explícito): " +
    fechaConHora.format(DateTimeFormatter.ISO_DATE_TIME));
DateTimeFormatter esDateFormat =
    DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm:ss");
System.out.println("Formato español (manual): " +
    fechaConHora.format(esDateFormat));
```

```
Formato por defecto: 2020-07-06T20:40:15
Formato ISO 8601 (explícito): 2020-07-06T20:40:15
Formato español (manual): 06/07/2020 08:40:15
```

Como vemos, existen diversos formateadores ya predefinidos en forma de constantes de `DateTimeFormatter`, como el que hemos usado (hay más que no se ven en la captura pero puedes consultar [aquí](#)):

```
LocalDateTime fechaConHora = LocalDateTime.parse("2020-07-06T20:40:15");
System.out.println(
//Parseando un form
LocalDate seisNov =
);
System.out.println(
//Formato personali
System.out.println(
ANSI_RESET);
System.out.println(
System.out.println(
(DateTimeFormatter.));
DateTimeFormatter esDateFormat = DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm:ss");
System.out.println("Formato por defecto: " + fechaConHora.format(DateTimeFormatter.ISO_DATE_TIME));
```

En el caso del formato manual hemos empleado los símbolos de formato que indicábamos en el apartado anterior para el *parsing*.

Hay que tener cuidado, no obstante, con el uso que se hace de alguno de ellos. Por ejemplo, supongamos que queremos que el mes se muestre con su nombre completo y no con un número; para ello usaríamos el formato "MMMM" (las 4 *M* representan el nombre del mes). Por defecto ese nombre se mostraría en inglés, por lo que veríamos por pantalla, por ejemplo "July" y no "Julio", en español, que es lo que nos interesa. ¿Cómo lo solventamos? **¿Cómo conseguimos localizar ese valor?**

Pues utilizando **el método `withLocale()`** de la clase `DateTimeFormatter`, que toma como argumento un objeto de la clase `java.util.Locale`.

Esta clase dispone [de algunas constantes](#) con valores predefinidos de idiomas, como el inglés, inglés americano, alemán... hasta francés canadiense, pero no español (se ve que es un idioma poco utilizado). Por ello, para nuestro idioma tendremos que definirlo explícitamente. El siguiente código te muestra cómo conseguirlo:

```
DateTimeFormatter esDateFormatLargo =  
    DateTimeFormatter  
        .ofPattern("EEEE, dd 'de' MMMM 'de' yyyy 'a las' hh:mm:ss")  
        .withLocale(new Locale("es", "ES"));  
System.out.println("Formato español (largo, localizado): " +  
    fechaConHora.format(esDateFormatLargo));
```

```
Formato español (largo, localizado): lunes, 06 de julio de 2020 a las 08:40:15
```

Fíjate en dos cosas:

- En primer lugar creamos el patrón que nos interesa usando `EEEE` para el nombre largo del día de la semana (mira la [tabla de formatos](#) de nuevo) y "escapeamos" todos los fragmentos que no son formato, como el "de" y el "a las" usando una comilla simple.
- Instanciamos un nuevo objeto `Locale` pasándole como parámetros el [código ISO 639](#) de idioma y el [código ISO 3166](#) de país. Así somos más

explícitos (español de España, pero podría haber sido español de México o de otro país hispanohablante). Si le hubiésemos pasado tan solo el primer parámetro funcionaría con la versión más neutra del idioma (simplemente español), que en esta ocasión no tendría diferencia alguna.

En este caso veríamos por pantalla la fecha formateada de la siguiente manera: `lunes, 06 de julio de 2020 a las 08:40:15`, que es lo que buscábamos.

Finalmente, nos faltaría saber cómo formatear la fecha con el formato actual del usuario de la aplicación, en lugar de uno arbitrario elegido por nosotros. Para ello usaremos la misma técnica, pero antes tenemos que averiguar el idioma y país del usuario actual. (He sangrado el código un poco para ganar en claridad y no verlo todo en una sola línea en la tercera sentencia):

```
String idiomaLocal = System.getProperty("user.language");
String paisLocal = System.getProperty("user.country");
System.out.println("Formato actual del sistema (" + idiomaLocal +
    "-" + paisLocal + "): " +
    fechaConHora.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT)
        .withLocale(
            new Locale(idiomaLocal, paisLocal)
        )
    )
);
```

```
Formato actual del sistema (en-US): 7/6/20, 8:40 PM
```

Usamos las propiedades del sistema para averiguar el código de idioma y el código de país, y luego procedemos de la misma manera que antes, con un nuevo `Locale` en función de estos datos.

La única cosa nueva que tenemos aquí es que, en lugar de definir el formato manualmente, hacemos uso del valor `SHORT` de la [enumeración](#) `FormatStyle` para expresar de manera genérica el formato

corto de fecha y hora. En este caso no podríamos haber utilizado el formato largo (`LONG`) ni completo (`FULL`) porque necesitan la información de zona horaria, que nuestra fecha de ejemplo no tiene por ser un `LocalDateTime` y no un `ZonedDateTime`.