# MANEJO DE FECHAS Y HORAS EN JAVA

El **manejo de fechas en Java** puede resultar en un principio un poco menos intuitivo que en otros lenguajes tales como PHP o Visual Basic. Trabajar con fechas en java no es algo del otro mundo ni tiene demasiadas complicaciones, pero la cantidad de formas que hay para hacerlo puede confundirnos, o peor aún, puede que sólo conozcamos la más mala para hacerlo. A continuación pretendo explicar que clases tiene el lenguaje para trabajar con fechas, los métodos más usados y algunas sugerencias para realizar un trabajo lo más correcto posible.

## 1.-. Clase Date

La clase ***Date*** del paquete java.util representa un instante de tiempo dado con precisión de milisegundos. La información sobre fecha y hora se almacena en un entero ***long*** de 64 bits, que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970 GMT (*Greenwich mean time*). Ya se verá que otras clases permiten a partir de un objeto ***Date*** obtener información del año, mes, día, hora, minuto y segundo.

El constructor por defecto ***Date()*** crea un objeto a partir de la fecha y hora actual del ordenador. El constructor ***Date(long)*** crea el objeto a partir de los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT. Los métodos ***after()*** y ***before()*** permiten saber si la fecha indicada como argumento implícito (***this***) es posterior o anterior a la pasada como argumento explícito. Los métodos ***getTime()*** y ***setTime()*** permiten obtener o establecer los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT para un determinado objeto ***Date***. Otros métodos son consecuencia de las interfaces implementadas por la clase ***Date***.

Los objetos de esta clase no se utilizan mucho directamente, entre otras cosas porque muchos de sus métodos están obsoletos, en su lugar se utilizan las clases que se van a ver a continuación.

## 2.- Clases Calendar y Gregorian Calendar

La clase ***Calendar*** es una clase ***abstract*** que dispone de métodos para convertir objetos de la clase ***Date*** en enteros que representan fechas y horas concretas. Nos permitirá modificar y obtener sus datos enteros tales como YEAR, MONTH, DAY, etc…

La clase ***GregorianCalendar*** es la única clase que deriva de ***Calendar*** y es la que se utilizará de ordinario.

***Java*** tiene una forma un poco particular para representar las fechas y horas:

1. Las horas se representan por enteros de 0 a 23 (la hora "0" va de las 00:00:00 hasta la 1:00:00), y los minutos y segundos por enteros entre 0 y 59.
2. Los días del mes se representan por enteros entre 1 y 31 (lógico).
3. Los meses del año se representan mediante enteros de 0 a 11 (no tan lógico).
4. Los años se representan mediante enteros de cuatro dígitos. Si se representan con dos dígitos, se resta 1900. Por ejemplo, con dos dígitos el año 2000 es para Java el año 00.

La clase ***Calendar*** tiene una serie de variables miembro y constantes (variables ***final***) que pueden resultar muy útiles:

- ✓ La variable ***int*** AM_PM puede tomar dos valores: las constantes enteras AM y PM.
- ✓ La variable ***int*** DAY_OF_WEEK puede tomar los valores ***int*** SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY y SATURDAY.
- ✓ La variable ***int*** MONTH puede tomar los valores ***int*** JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER. Para

hacer los programas más legibles es preferible utilizar estas constantes simbólicas que los correspondientes números del 0 al 11.

- ✓ La variable miembro HOUR se utiliza en los métodos ***get()*** y ***set()*** para indicar la hora de la mañana o de la tarde (en relojes de 12 horas, de 0 a 11). La variable HOUR_OF_DAY sirve para indicar la hora del día en relojes de 24 horas (de 0 a 23).
- ✓ Las variables DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_MONTH (o bien DATE), DAY_OF_YEAR, WEEK_OF_MONTH, WEEK_OF_YEAR tienen un significado evidente.
- ✓ Las variables ERA, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND tienen también un significado evidente.
- ✓ Las variables ZONE_OFFSET y DST_OFFSET indican la zona horaria y el desfase en milisegundos respecto a la zona GMT.

La clase ***Calendar*** dispone de un gran número de métodos para establecer u obtener los distintos valores de la fecha y/u hora.

La clase ***GregorianCalendar*** añade las constante BC y AD para la ERA, que representan respectivamente antes y después de Jesucristo. Añade, además, varios constructores que admiten como argumentos la información correspondiente a la fecha/hora y, opcionalmente, la zona horaria.

Para instanciar un objeto de clase **Calendar**, tendremos 2 opciones:

- **Opción A**: Utilizar el método **getInstance()**, que nos proporcionará un objeto **Calendar** cuyos campos han sido inicializados con la fecha y la hora actuales, con base a la zona horaria.

```
Calendar c = GregorianCalendar.getInstance();
```

- **Opción B**: Utilizar el constructor de la clase **GregorianCalendar**. Esta clase es a su vez una subclase de **java.util.Calendar**, que nos proporciona un calendario estándar usado comúnmente en la mayor parte del mundo.

```
Calendar c = new GregorianCalendar();
```

A continuación se muestra un ejemplo de utilización de estas clases. Os sugiero que creéis y ejecutéis el siguiente programa, observando los resultados impresos en la consola.

```
import java.util.*;

public class PruebaFechas {
   public static void main(String arg[]) {

         Date d = new Date();
         GregorianCalendar gc = new GregorianCalendar();
         gc.setTime(d);

         Calendar calendar1 = Calendar.getInstance();
         Calendar calendar2 = new GregorianCalendar();

         System.out.println("Variable Calendar1 (getInstance): "
                     +calendar1.get(Calendar.DAY_OF_MONTH)+"/"
                     +calendar1.get(Calendar.MONTH)+"/"
                     +calendar1.get(Calendar.YEAR));

         System.out.println("VariabLe Calendar2 (new Gregorian Calendar): "
                     +calendar2.get(Calendar.DAY_OF_MONTH)+"/"
                     +calendar2.get(Calendar.MONTH)+"/"
                     +calendar2.get(Calendar.YEAR));

         System.out.println("--------------");
         System.out.println("Era: "+gc.get(Calendar.ERA));
```

```
        System.out.println("Year: "+gc.get(Calendar.YEAR));
        System.out.println("Month: "+gc.get(Calendar.MONTH));
        System.out.println("Dia del mes: "+gc.get(Calendar.DAY_OF_MONTH));
        System.out.println("D de la S en mes:"
                           +gc.get(Calendar.DAY_OF_WEEK_IN_MONTH));
        System.out.println("No de semana: "+gc.get(Calendar.WEEK_OF_YEAR));
        System.out.println("Semana del mes: "+gc.get(Calendar.WEEK_OF_MONTH));
        System.out.println("Fecha: "+gc.get(Calendar.DATE));
        System.out.println("Hora: "+gc.get(Calendar.HOUR));
        System.out.println("Tiempo del dia: "+gc.get(Calendar.AM_PM));
        System.out.println("Hora del dia: "+gc.get(Calendar.HOUR_OF_DAY));
        System.out.println("Minuto: "+gc.get(Calendar.MINUTE));
        System.out.println("Segundo: "+gc.get(Calendar.SECOND));
        System.out.println("Dif. horaria: "+gc.get(Calendar.ZONE_OFFSET));
        }
}
```

Veamos otro ejemplo para ir aclarándonos. Mi cumpleaños es en octubre, el día 27, y deseo saber qué día lo celebré en el 2010; para eso obtengo una instancia de Calendar (que siempre devuelve un objeto del tipo GregorianCalendar) y la ajusto al 27 de octubre de 2010, luego obtengo el nombre del día, veamos:

```
Calendar cumpleCal = Calendar.getInstance();
cumpleCal.set(2010,9,27);
   //La hora no me interesa y recuerda que los meses van de 0 a 11
int dia = cumpleCal.get(Calendar.DAY_OF_WEEK);
System.out.println(dia); //Día 4 = WEDNESDAY = MIÉRCOLES
```

## *¿Cómo modificar los atributos o valores de la fecha?*

Al igual que disponemos del método **get(int field)** para obtener cualquier valor de una fecha, disponemos también del método **set(int field, int value)** para modificarlo. Deberemos pasar 2 parámetros:

- *field:* atributo de la fecha que queremos modificar
- *value:* nuevo valor a asignar

Veamos a continuación un ejemplo de cómo modificar los atributos año, mes y día:

```
calendar1.set(Calendar.YEAR, 2020);
calendar1.set(Calendar.MONTH, 02); // OJO: Recuerda que los valores de los meses
                                   // comienzan por 0.
calendar1.set(Calendar.DATE, 30);

// Cambiar el mes a enero
calendar1.set(Calendar.MONTH,Calendar.JANUARY)
```

Además del método simple **set(int field, int value)**, disponemos también de 3 métodos más sobrecargados **set** que nos permitirán **asignar fechas completas** en un único paso:

```
// Asignamos año, mes y día.
calendar1.set(2020, 02, 30);
// Asignamos año, mes, día, horas y minutos.
calendar1set(2020, 02, 30, 19, 00);
// Asignamos año, mes, día, horas, minutos y segundos.
calendar1.set(2020, 02, 30, 19, 00, 55);
```

## *¿Cómo sumar y restar días a fechas?*

Realizar operaciones como sumar o restar días no es algo que dependa directamente de Calendar sino más bien de una subclase de esta que implemente algún tipo de calendario usado, pues no todos los calendarios tienen 12 meses ni años de 365 días como el que nosotros (en casi todo occidente) usamos. Este calendario usado en occidente, llamado gregoriano fue adoptado por primera vez en 1582 por el imperio

romano (o aproximadamente) y posteriormente se fue adoptando en muchos otros países, por ejemplo 1752 en Gran Bretaña y 1918 en Rusia.

A grandes rasgos sabemos que el calendario gregoriano consta de años que son definidos por cada traslación (vuelta de la tierra alrededor del sol), cada año tiene doce meses de los cuales 7 tienen 31 días, 4 tienen 30 días y 1 tiene 28 días excepto en años bisiestos que tiene 29. Estos años bisiestos se implantaron para corregir el desfase que tenemos cada cuatro años (un año real dura 365 días y 6 horas aproximadamente), regla completa para los años bisiestos según el calendario gregoriano es la siguiente:

*"Un año es bisiesto si es divisible por 4, a menos que sea divisible por 100 y no por 400"*.

Todo esto quizá suene un poco tonto, pero es absolutamente necesario tenerlo claro para entender el funcionamiento de la clase GregorianCalendar.

GregorianCalendar es una subclase de Calendar y es la implementación directa del calendario tal y como lo conocemos hoy día. Es con esta clase con la que podemos sumar 2 ó 3 días a una fecha sin preocuparnos por desbordamientos o recalcular meses o años, pues ella lo hace automáticamente tomando en cuenta las reglas en los párrafos anteriores.

He aquí uno de los métodos más útiles cuando queremos **operar con fechas**: **add(int field, int amount)**. Con la misma estructura que el método **set(int field, int amount)**, podremos **sumar** (y por lo tanto **restar**) cantidades de diferentes valores a una **fecha**.

- *field:* atributo de la fecha que queremos modificar y cuya cantidad pasaremos en el parámetro amount
- *amount:* cantidad a sumar o restar (valores negativos) a la fecha

Por ejemplo:

```
// Sumamos 30 minutos a la fecha actual.
calendar.add(Calendar.MINUTE, 30);
// Sumamos 100 días a la fecha actual.
calendar.add(Calendar.DATE, 100);
// Restamos 10 años a la fecha actual.
calendar.add(Calendar.YEAR, -10);
```

Otro método muy importante que hay que conocer sobre la clase Calendar es el método **roll(),** este método hace lo mismo que add(), la diferencia es que con add(), si llegamos a aumentar 12 horas al calendario, y eran las 16:00 del 20 de agosto, la fecha se recorre a 04:00 del 21 de agosto, ahora si usamos el método roll(), la fecha quedaria solo como 04:00 del 20 de agosto.

Por ejemplo:

```
// asumamos que estamos a Octubre 8, 2001, y que c es una
// instancia de un calendario con esa fecha
c.roll(Calendar.MONTH, 9); // aumentamos 9 meses
Date d2 = c.getTime();
System.out.println(”nueva fecha ” + d2.toString() ; // observar el año en la salida!
```

Daría una salida similar a esta:

```
nueva fecha Fri Jul 08 19:46:40 MDT 2001
```

Observamos que el mes cambio de octubre a julio, pero el año es el mismo.

## *¿Cómo calcular los días entre dos fechas?*

Para calcular la cantidad de días entre dos fechas con el API de Java (java.util) debería hacer lo siguiente:

```
//Creo las dos instancias de fecha
GregorianCalendar gc = new GregorianCalendar(2000, 11, 20);
GregorianCalendar gc1 = new GregorianCalendar(2000, 11, 25);
```

```
//Obtengo los objetos Date para cada una de ellas
Date fec1 = gc.getTime();
Date fec2 = gc1.getTime();

//Realizo la operación
long time = fec2.getTime() - fec1.getTime();

//Muestro el resultado en días
System.out.println(time/(3600*24*1000));
```

Ahora, un par de consideraciones:

- El método **getTime()** retorna un long que simboliza la cantidad de milisegundos transcurridos desde el 01/01/1970.
- El resultado de restar los dos long da como resultado (nuevamente) la cantidad de milisegundos.
- La fórmula **time/(3600*24*1000)** sirve para pasar los milisegundos a días.

## 3.- Clases DateFormat y SimpleDateFormat

***DateFormat*** es una clase ***abstract*** que pertenece al package ***java.text*** y no al package ***java.util***, como las vistas anteriormente. La razón es para facilitar todo lo referente a la ***internacionalización***, que es un aspecto muy importante en relación con la conversión, que permite dar formato a fechas y horas de acuerdo con distintos criterios locales. Esta clase dispone de métodos ***static*** para convertir ***Strings*** representando fechas y horas en objetos de la clase ***Date***, y viceversa.

Veamos el siguiente ejemplo:

```
1. import java.text.*;
2. import java.util.*;
3. class Fecha2 {
4.   public static void main(String[] args) {
5.     Date d1 = new Date(1000000000000L);//usamos el constructor basado en milisegun
   dos, le pasamos un trillón de milisegundos
6.
7.     DateFormat[] dfa = new DateFormat[6];
8.     dfa[0] = DateFormat.getInstance();
9.     dfa[1] = DateFormat.getDateInstance();
10.    dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
11.    dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
12.    dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
13.    dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);
14.    for(DateFormat df : dfa) //recorremos el array y mostramos la fecha con los di
   stintos formatos
15.      System.out.println(df.format(d1));
16.  }
17.}
```

El código anterior produce la siguiente salida en mi máquina:

```
9/09/01 3:46
09-sep-2001
9/09/01
09-sep-2001
9 de septiembre de 2001
domingo 9 de septiembre de 2001
```

Examinando este código vemos unas cuantas cosas. Por un lado, DateFormat es otra clase abstracta, así que no podemos usar *new* para crear instancias de DateFormat. En este caso usamos 2 métodos, getInstance() y getDateInstance().

También usamos campos estáticos de la clase DateFormat para personalizar nuestras instancias de DateFormat. Cada uno de estos campos estáticos representan un estilo de formato.

Podríamos deducir que la versión sin argumentos del método getDateInstance() nos da el mismo estilo que el método getDateInstance(DateFormat.MEDIUM) de la clase DateFormat, pero eso no siempre es así. Otro método con el cual deberemos familiarizarnos es el método parse(), quién toma una String de fecha formateada con algún estilo de DateFormat y convierte la String en un objeto Date. Este método lanza una excepción(ParseException) en caso de que se le haya pasado un String mal formateado.

El siguiente código crea una instancia de Date, usa DateFormat.format() para pasar Date a String y luego usa parse() para regresar el String a Date:

```
1.  import java.text.DateFormat;
2.  import java.text.ParseException;
3.  import java.util.*;
4.
5.  class Fechas3 {
6.
7.   public static void main(String[] args) {
8.         Date d1 = new Date(10000000000000L);
9.         System.out.println("fecha 1= " + d1.toString());
10.
11.        DateFormat df = DateFormat.getDateInstance(
12.        DateFormat.SHORT);
13.        String s = df.format(d1);
14.        System.out.println(s);
15.        try {
16.              Date d2 = df.parse(s);
17.              System.out.println("fecha parseada = " + d2.toString());
18.        } catch (ParseException pe) {
19.              System.out.println("error de parseo");
20.        }
21.  }
22. }
```

El código de arriba produce en mi máquina:

```
fecha 1= Sun Sep 09 03:46:40 CEST 2001
9/09/01
fecha parseada = Sun Sep 09 00:00:00 CEST 2001
```

Hay que observar que perdimos precisión a la hora de parsear la fecha, esto se debe a que la formateamos con el estilo DateFormat.SHORT.

La clase ***SimpleDateFormat*** es la única clase derivada de ***DateFormat***. Es la clase que conviene utilizar, ya que nos permitirá dar formato a fechas y horas para mostrarlas al usuario final. Esta clase se utiliza de la siguiente forma: se le pasa al constructor un ***String*** definiendo el formato que se desea utilizar, teniendo en cuenta la siguiente tabla de la siguiente página.

Por ejemplo:

```
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
```

Este ejemplo al formatear mediante el método ***format*** una fecha daría como resultado algo así: 2010/07/24. format recibe un objeto Date el cual obtendremos de nuestra instancia de GregorianCalendar a través del método *getTime* el cual nos retorna un Date que representa el instante de tiempo de nuestra fecha. Por ejemplo:

```
System.out.println(dateFormat.format(c.getTime()));
```

| Letter | Date or Time Component | Presentation | Examples |
|---|---|---|---|
| G | Era designator | Text | AD |
| y | Year | Year | 1996; 96 |
| M | Month in year | Month | July; Jul; 07 |
| w | Week in year | Number | 27 |
| W | Week in month | Number | 2 |
| D | Day in year | Number | 189 |
| d | Day in month | Number | 10 |
| F | Day of week in month | Number | 2 |
| E | Day in week | Text | Tuesday; Tue |
| a | Am/pm marker | Text | PM |
| H | Hour in day (0-23) | Number | 0 |
| k | Hour in day (1-24) | Number | 24 |
| K | Hour in am/pm (0-11) | Number | 0 |
| h | Hour in am/pm (1-12) | Number | 12 |
| m | Minute in hour | Number | 30 |
| s | Second in minute | Number | 55 |
| S | Millisecond | Number | 978 |
| z | Time zone | General time zone | Pacific Standard Time; PST; GMT-08:00 |
| Z | Time zone | RFC 822 time zone | -0800 |

La clase **SimpleDateFormat** también nos permite obtener un objeto Date a través de una cadena de texto así:

```
try {
    Date d = dateFormat.parse("2010/07/24");
    System.out.println(dateFormat.format(d));
}
catch (ParseException ex) {
   ex.printStackTrace();
}
```

Como observamos este método lanza una Excepción de tipo **java.text.ParseException** en caso de que la cadena no represente una fecha valida.

Veamos otro ejemplo completo:

```
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
class PruebaFechas2
{
  public static void main(String[] args)
  {
        Date hoy = new Date();
        //
        // Primera prueba, almacena en "salida", la fecha y hora actuales
        // según el formato del patron especificado
        String patron = "EEEE dd-MMM-yyyy, HH:mm:ss";
        SimpleDateFormat formato = new SimpleDateFormat(patron);
        String salida = formato.format(hoy);
        System.out.println("Fecha: " + salida);
        //
        // Segunda prueba, almacenan en sFecha y sHora la fecha y hora según el
        // formato local predeterminado
        String sFecha, sHora;
        DateFormat formato1;
        formato1 = DateFormat.getDateInstance();
```

```
            sFecha = formato1.format(hoy);
            formato1 = DateFormat.getTimeInstance();
            sHora = formato1.format(hoy);
            System.out.println("Fecha: " + sFecha);
            System.out.println("Hora: " + sHora);
        }
    }
```

La salida de este programa podría ser:

```
Fecha: martes 05-abr-2011, 21:46:30
Fecha: 05-abr-2011
Hora: 21:46:30
```

La documentación de la clase ***SimpleDateFormat*** proporciona abundante información al respecto, incluyendo algunos ejemplos.

### *¿Cómo validar fechas?*

A través de los ejemplos que se han ido viendo hasta ahora, ya tendremos una idea de cómo se podría validar una fecha, no obstante a continuación se muestra un método que permite la validación de fechas, a partir de un String.

```
public static boolean esFechaCorrecta(String fecha)
{
   try {
       SimpleDateFormat formatoFecha =
              new SimpleDateFormat("dd/MM/yyyy",Locale.getDefault());

       // El método setLenient a false obliga a que la fecha
       // "tenga sentido estricto", y por lo tanto rechaza un
       // "30 de febrero" o un "29 de febrero de 2007" como fechas válidas.
       //    Si no establecemos el lenient a false, al parsear una fecha
       // "interpretará" la fecha correcta. Un "30 de febrero" se convertirá
       // en 1 marzo, (en 2 de marzo si es un año no bisiesto)...
       formatoFecha.setLenient(false);

       // El método parse devuelve un objeto Date, por tanto si el String que
       // le llega no es una fecha correcta, bien por formato (Ej: 12/hola),
       // bien porque el día, mes o año sean incorrectos (Ej: 30/02/2011)
       // lanza una excepción del tipo ParseException
       formatoFecha.parse(fecha);
       }
    catch (ParseException e) {
         return false;
    }

    return true;
}
```

El anterior método nos devolvería *true* para llamadas cuyo argumento sea “28/12/2012”, o “15/03/1987”, y false para llamadas con argumentos cómo “29/02/2011”, “31/04/2020”, “23/hola”, etc.

## 4.- La clase Locale

Esta clase es usada por DateFormat y NumberFormat para personalizar el formato de fechas y números con respecto a un Locale específico.

Esta clase tiene 2 constructores:

```
Locale(String lenguaje)
Locale(String lenguaje, String país)
```

El argumento de lenguaje representa un código ISO 639:

| Código de lenguaje | Descripción |
|---|---|
| **de** | Alemán |
| **en** | Inglés |
| **fr** | Francés |
| **ja** | Japonés |
| **es** | Español |
| **ko** | Coreano |
| **zh** | Chino |

Por ejemplo si queremos manejar italiano en nuestra aplicación, sólo necesitamos el código del lenguaje, pero si queremos representar el italiano que se usa en Suiza, necesitamos usar el segundo constructor pasándole el código de país de suiza “CH”

```
Locale locPT = new Locale("it");        // Italiano
Locale locBR = new Locale("it", "CH");  // Suiza
```

Esto nos daría una salida parecida a esto:

```
sabato 1 ottobre 2005
sabato, 1. ottobre 2005
```

Ahora hay que poner todo junto y hacer un código que crea un objeto de Calendar, le cambiamos la fecha a otro día cualquiera y luego lo mandamos a un objeto Date, para al final formatear esa fecha con diferentes Locales.

```
import java.text.DateFormat;
import java.util.*;

class Fechas4 {
  public static void main(String[] args)
  {
        Calendar c = Calendar.getInstance();
        c.set(1989, 1, 21); // Febrero 21, 1989
        // el  mes se basa en 0, osea Enero = 0
        // esa fecha es el día que nací

        Date d = c.getTime();
        Locale locIT = new Locale("it", "IT");  // Italy
        Locale locPT = new Locale("pt");        // Portugal
        Locale locBR = new Locale("pt", "BR");  // Brazil
        Locale locIN = new Locale("hi", "IN");  // India
        Locale locMEX = new Locale("es", "MX"); // México

        DateFormat dfUS = DateFormat.getInstance();
        System.out.println("US       " + dfUS.format(d));
        DateFormat dfUSfull = DateFormat.getDateInstance(
        DateFormat.FULL);
        System.out.println("US largo " + dfUSfull.format(d));

        DateFormat dfIT = DateFormat.getDateInstance(
        DateFormat.FULL, locIT);
        System.out.println("Italy    " + dfIT.format(d));
        DateFormat dfPT = DateFormat.getDateInstance(
        DateFormat.FULL, locPT);
        System.out.println("Portugal " + dfPT.format(d));
        DateFormat dfBR = DateFormat.getDateInstance(
        DateFormat.FULL, locBR);
        System.out.println("Brazil   " + dfBR.format(d));
        DateFormat dfIN = DateFormat.getDateInstance(
        DateFormat.FULL, locIN);
        System.out.println("India    " + dfIN.format(d));
        DateFormat dfMX = DateFormat.getDateInstance(
```

```
        DateFormat.FULL, locMEX);
        System.out.println("México   " + dfMX.format(d));
 }
}
```

En mi JVM produce la siguiente salida:

```
US       21/02/89 17:23
US largo martes 21 de febrero de 1989
Italy    martedì 21 febbraio 1989
Portugal Terça-feira, 21 de Fevereiro de 1989
Brazil   Terça-feira, 21 de Fevereiro de 1989
India    ???????, ?? ??????, ????
México   martes 21 de febrero de 1989
```

## 5.- Clases TimeZone y SimpleTimeZone

La clase ***TimeZone*** es también una clase ***abstract*** que sirve para definir la zona horaria. Los métodos de esta clase son capaces de tener en cuenta el cambio de la hora en verano para ahorrar energía. La clase ***SimpleTimeZone*** deriva de ***TimeZone*** y es la que conviene utilizar.

El valor por defecto de la zona horaria es el definido en el ordenador en que se ejecuta el programa. Los objetos de esta clase pueden ser utilizados con los constructores y algunos métodos de la clase ***Calendar*** para establecer la zona horaria.

# Anexo I: La clase java.sql.Date

**java.sql.Date**: Esta clase hereda de java.util.Date y es la representación de la fecha cuando trabajamos con JDBC (Java DabaBase Connectivity), es decir, son los campos almacenados en una base de datos cuyo tipo es una fecha que puede o no incluir la hora, aunque la clase java.sql.Date siempre lo hace. Al igual que su clase padre, tiene una precisión de milisegundos, con la excepción que al mostrarla en la salida estándar con el formato por defecto solo muestra el día, mes y año. Hay que anotar también que para campos que almacenen solamente horas existen otras clases para manejarlos.

En resumen ambas clases, sólo se encargan de almacenar la cantidad de milisegundos que han pasado desde las 12 de la noche del primero de enero de 1970 en el meridiano de Greenwich. Aquí vienen dos puntos importantes:
a) Si la fecha que almacena cualquiera de las clases es menor a las 00:00:00 enero 1 de 1970 GMT, su valor el milisegundos será negativo.
b) La fecha es susceptible a la zona horaria. Por ejemplo en Colombia los milisegundos no se empiezan a contar desde enero 1 de 1970, sino a partir de las 19:00 de diciembre 31 de 1969. Esto es importante por que si transportamos una fecha relativa de una zona a otra, podemos llegar a tener problemas al confiar en los milisegundos que se tienen; además como la clase intenta representar el "Tiempo Universal Coordinado" (UTC) suma 0.9 segundos cada año para ajustar la diferencia entre el reloj atómico y la velocidad de rotación de la tierra. Esto se traduce en que muy dificilmente podemos basarnos en valores como 0 o 60000 para realizar validaciones, pues esos milisegundos no son controlables cuando creamos la instancia de una fecha, peor aún, los milisegundos no son ni siquiera iguales para la misma fecha en la misma zona horaria.

Ambas clases se pueden instanciar directamente mediante **new()**, pero la clase **java.sql.Date** necesita un parámetro en el constructor: el tiempo en milisegundos, así que las siguientes instrucciones son válidas:

```
java.util.Date fechaActual = new java.util.Date(); //Fecha actual del sistema
java.sql.Date inicioLocal = new java.sql.Date(0); //Milisegundo cero

//también se puede crear una instancia de java.util.Date con parámetros iniciales
java.util.Date otraFecha = new java.util.Date(1000); //El primer segundo a partir del
inicio
```

Prueba a imprimir cada uno de estos valores y fíjate en la diferencia de formatos entre java.sql.Date y java.util.Date. Se puede pasar de java.sql.Date a java.util.Date de dos fomas, una de ellas es con una asignación simple:

```
java.util.Date utilDate = null;
java.sql.Date sqlDate = new java.sql.Date(0);
utilDate = sqlDate;
/* aunque es java.util.Date,
 si la imprimes tendrá el formato de java.sql.Date, recordemos que java.sql.Date hereda
de
 java.util.Date */
System.out.println(utilDate);
```

También se pueden tomar los milisegundos de java.sql.Date y pasarlos al constructor de java.util.Date:

```
java.util.Date utilDate = null;
java.sql.Date sqlDate = new java.sql.Date(0);
utilDate = new java.util.Date(sqlDate.getTime());
//esta vez se mostrará con el formato de java.util.Date
System.out.println(utilDate);
```

Para pasar de java.util.Date a java.sql.Date se deben tomar los milisegundos de la primera y pasarlos al constructor de la segunda:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
```

```
//Con formato de java.sql.Date
System.out.println(sqlDate);
```

Para comparar fechas usamos el método compareTo() que internamente compara los milisegundos entre ellas usando directamente los métodos getTime() de ambas clases.

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
if (utilDate.compareTo(sqlDate) == 0){
   System.out.println("IGUALES");
}else{
   System.out.println("DIFERENTES");
}
```

O lo que es equivalente:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
if (utilDate.getTime() == sqlDate.getTime()){
  System.out.println("IGUALES");
}else{
  System.out.println("DIFERENTES");
}
```

## 2. Las clases Time y Timestamp.

Ambas clases pertenecen al API JDBC y son la encargadas de representar los campos de estos tipos en una base de datos. Esto no quiere decir que no se puedan usar con otros fines. Al igual que java.sql.Date, son hijas (heredan) de java.util.Date, es decir, su núcleo son los milisegundos.
La clase Time es un envoltorio de la clase java.util.Date para representar los datos que consisten de horas, minutos, segundos y milisegundos, mientras Timestamp representa estos mísmos datos más un atributo con nanosegundos, de acuerdo a las especificaciones del lenguaje SQL para campos de tipo TIMESTAMP.
Como ambas clases heredan del java.util.Date, es muy fácil pasar de un tipo de dato a otro; similar a la clase java.sql.Date, tanto Time como Timestamp se pueden instanciar directamente y su constructor tiene como parámetro el número de milisegundos; como es de imaginarse, cuando se muestra alguna de las clases mediante su método toString() se ven los datos que intentan representar; La clase Time sólamente muestra la hora, minutos y segundo, mientras timestamp agrega fracciones de segundo a la cadena.
Para convertir entre tipos de datos diferentes debemos usar los milisegundos de una clase y asignarlos a las instancias de las otras, y como la clase java.util.Date es superclase de todas, a una instancia de esta podemos asignar cualquiera de las otras, manteniendo los métodos de la clase asignada, es decir, si asignamos un Time a una java.util.Date, al imprimir se verá el mismo formato de la clase Time.
Con este código:

```
  java.util.Date utilDate = new java.util.Date(); //fecha actual
  long lnMilisegundos = utilDate.getTime();
  java.sql.Date sqlDate = new java.sql.Date(lnMilisegundos);
  java.sql.Time sqlTime = new java.sql.Time(lnMilisegundos);
  java.sql.Timestamp sqlTimestamp = new java.sql.Timestamp(lnMilisegundos);
  System.out.println("util.Date: "+utilDate);
  System.out.println("sql.Date: "+sqlDate);
  System.out.println("sql.Time: "+sqlTime);
  System.out.println("sql.Timestamp: "+sqlTimestamp);
```

Se obtiene la siguiente salida:

```
util.Date: Thu May 20 19:01:46 GMT-05:00 2004
sql.Date: 2004-05-20
sql.Time: 19:01:46
sql.Timestamp: 2004-05-20 19:01:46.593
```

Note que aún cuando todos los objetos tienen los mismos milisegundos el formato con el que se muestran dependen de la clase que realmente los contiene. Es decir, no importa que a un objeto del tipo java.util.Date se le asigne uno del tipo Time, al mostrar a través de la consola se invocará el método toString() de la clase time:

```
utilDate = sqlTime;
System.out.println("util.Date apuntando a sql.Time: ["+sqlTime+"]");
utilDate = sqlTimestamp;
System.out.println("util.Date apuntando a sql.Timestamp: ["+sqlTimestamp+"]");
```

Arroja:

```
util.Date apuntando a sql.Time: [19:29:47]
util.Date apuntando a sql.Timestamp: [2004-05-20 19:29:47.468]
```

Pero si en vez de solo apuntar, creamos nuevas instancias con los milisegundos los formatos con que se muestran son los mismos. Note que lo verdaderamente importante ocurre cuando creamos la instancia de java.util.Date usando los milisegundos del objeto sqlTime, pues aunque este último únicamente muestra horas, minutos y segundos, siempre ha conservado todos los datos de la fecha con que se creó.

```
utilDate = new java.util.Date(sqlTime.getTime());
System.out.println("util.Date con milisegundos de sql.Time: ["+utilDate+"]");
utilDate = new java.util.Date(sqlTimestamp.getTime());
System.out.println("util.Date con milisegundos de sql.Timestamp: ["+utilDate+"]");
```

Fíjese en el formato de salida:

```
util.Date con milisegundos de sql.Time: [Thu May 20 19:54:42 GMT-05:00 2004]
util.Date con milisegundos de sql.Timestamp: [Thu May 20 19:54:42 GMT-05:00 2004]
```

Para finalizar esta primera entrega veamos el código para mostrar la diferencia entre dos fechas en horas, minutos y segundos. Esta no es la mejor forma para hacerlo, pero cabe bien para mostrar de forma práctica todos los conceptos anteriormente estudiados.

```
 import java.util.HashMap;
 import java.util.Map;
 public class Prueba {
  public static Map getDiferencia(java.util.Date fecha1, java.util.Date fecha2){
   java.util.Date fechaMayor = null;
   java.util.Date fechaMenor = null;
   Map resultadoMap = new HashMap();

   /* Verificamos cual es la mayor de las dos fechas, para no tener sorpresas al momento
    * de realizar la resta.
    */
   if (fecha1.compareTo(fecha2) > 0){
    fechaMayor = fecha1;
    fechaMenor = fecha2;
   }else{
    fechaMayor = fecha2;
    fechaMenor = fecha1;
   }

  //los milisegundos
   long diferenciaMils = fechaMayor.getTime() - fechaMenor.getTime();

   //obtenemos los segundos
   long segundos = diferenciaMils / 1000;

   //obtenemos las horas
   long horas = segundos / 3600;

   //restamos las horas para continuar con minutos
   segundos -= horas*3600;
```

```
   //igual que el paso anterior
   long minutos = segundos /60;
   segundos -= minutos*60;

   //ponemos los resultados en un mapa :-)
   resultadoMap.put("horas",Long.toString(horas));
   resultadoMap.put("minutos",Long.toString(minutos));
   resultadoMap.put("segundos",Long.toString(segundos));
   return resultadoMap;
}

  public static void main(String[] args) {
   //5:30:00 de Noviembre 10 - 1950 GMT-05:00
   java.util.Date fecha1 = new java.util.Date(-604070999750L);

   //6:45:20 de Noviembre 10 - 1950 GMT-05:00
   java.util.Date fecha2 = new java.util.Date(-604066478813L);

   //Luego vemos como obtuve esas fechas
   System.out.println(getDiferencia(fecha1, fecha2));
  }
}
```

```
DIFERENCIA ENTRE FECHAS
Fecha1: Fri Nov 10 05:30:00 GMT-05:00 1950
Fecha2: Fri Nov 10 06:45:21 GMT-05:00 1950
{segundos=20, horas=1, minutos=15}
```

Notas:
1. Existe un error de un segundo, lo cual no sucede cuando trabajamos con fechas posteriores a 1970, ¿Por qué?.
2. Este procedimiento funciona igual para todos los hijos de java.util.Date: java.sql.Date, java.util.Time y java.util.Timestamp.
3. Todos los ejemplos los hemos hecho creando nuevas instancias de las clases. He omitido el traer información desde una base de datos para no complicar el código; pero todo lo que hemos hecho debe funcionar igual de ambas formas (con base de datos y usando constructores).

# Anexo II: Introducción a Joda Time

## 1. Introducción

Joda Time es un API Java que permite trabajar con fechas de una forma más sencilla, potente y eficiente que el API estándar de fechas de Java. Joda Time incluye algunos conceptos como intervalos, duraciones y períodos, que están bastante mal soportados en el API estándar.

Joda Time es bastante "viejo", lleva desarrollándose al menos desde el 2002 y la última actualización es del año pasado, pero muchos desarrolladores lo desconocen y a veces resulta muy útil para realizar ciertas operaciones complejas con fechas (complejas si no utilizamos este API, claro).

Las características principales de Joda Time son:

- Facilidad de uso, con métodos de acceso directos a los campos de una fecha.
- Facilidad de extensión. Extender la clase Calendar del JDK puede resultar muy complicado si necesitásemos utilizar un sistema de calendario personalizado. Joda-Time soporta múltiples calendarios (8 actualmente) por medio de un sistema extensible basado en la clase Chronology. A pesar de todo, la mayoría de los mortales probablemente nunca necesitemos extender ninguno de los dos sitemas en nuestro trabajo cotidiano.
- Funcionalidades avanzadas para el cálculo y formateo de fechas, que en muchos casos son difíciles de imitar con el API estándar de Java.
- Es muy fácil la conversión de fechas entre Joda Time y el JDK.
- Una documentación bastante buena del uso general del API y un javadoc detallado que nos permiten utilizar funcionalidades avanzadas.
- Integración con otros sistemas como Hibernate o los tags de JSP.
- Es Open Source, bajo la licencia Apache License Version 2.0.

## 2. Configurar nuestra aplicación para utilizar Joda Time

Para utilizar el API de Joda Time en nuestra aplicación sólamente tendremos que descargarnos el .jar correspondiente (que ocupa unos 2 MB) y añadirlo a nuestro classpath. O, mejor aún, si estamos utilizando Maven para gestionar nuestro proyecto, tendremos que añadir la dependencia siguiente a nuestro "pom.xml":

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>1.6</version>
</dependency>
```

## 3. Conceptos clave

| Concepto | Descripción | Interfaz / Clase Abstracta | Implementaciones principales |
|---|---|---|---|
| Instante | Representa un instante concreto en la línea temporal (una fecha), con precisión de milisegundos.<br>Un instante se representa internamente como el número de milisegundos transcurridos desde "1970-01-01T00:00Z" | ReadableInstant | DateTime<br>MutableDateTime<br>Instant |
| Intervalo | Representa un intervalo de tiempo entre dos instantes que utilizan la misma cronología y zona horaria.<br>Los intervalos son abiertos por la derecha, es decir, incluyen su instante inicial, pero no el final. | ReadableInterval | Interval<br>MutableInterval |
| Duración | Representa un período de tiempo en milisegundos. No dependen de una cronología ni zona horaria. | ReadableDuration | Duration |
| Período | Representa un período de tiempo en términos de años, meses, semanas, días, horas, minutos, segundos y milisegundos. | ReadablePeriod | Period<br>MutablePeriod<br>Years |

| | | | |
|---|---|---|---|
| | Se diferencia de una duración en que no es exacto en términos de milisegundos, por ejemplo, sumar un período de "1 mes" al día "16 de Febrero" devolverá "16 de Marzo", que no es lo mismo que sumar 30 días o $2.592x10^6$ milisegundos.<br>Hay períodos de un solo campo (Years, Days...) o de varios cualesquiera (Period). | | Months<br>Days<br>Hours<br>Minutes |
| Parcial | Representa una fecha de forma incompleta y sin ninguna zona horaria.<br>Por ejemplo, un parcial podría representar el día "7 de Julio" (sin especificar el año), las "12:00" o "Enero de 1994". | ReadablePartial | LocalDate<br>LocalTime<br>LocalDateTime<br>Partial |
| Cronología | Representa un sistema de calendario/medición del tiempo, como la clase Calendar del JDK.<br>Para la mayoría de las aplicaciones, no necesitaremos utilizar estas clases, y nos valdrá con utilizar la implementación por defecto: ISOChronology. | Chronology | ISOChronology<br>GregorianChronology<br>JulianChronology |
| Zona horaria | Una zona horaria se aplica con el patrón Decorator sobre una cronología. | DateTimeZone (clase abstracta) | No se usan directamente, sino a través de métodos factoría de DateTimeZone |

# 4. Trabajar con fechas

Para todos los ejemplos que siguen a partir de ahora, supondremos que nuestro Locale es "es-ES".

## 4.1. Construcción de fechas

```
// Fecha y hora actuales
DateTime date = new DateTime();

// Especificar una fecha en formato ISO
date = new DateTime("2010-06-25"); // horas, minutos, segundos y milisegundos a 0
date = new DateTime("2010-06-25T13:30:00"); // milisegundos a 0

// Especificar indicando año, mes, día, horas, minutos, segundos y milisegundos
date = new DateTime(2010, 6, 25, 13, 30, 0, 0);

// Especificar zona horaria
date = new DateTime(DateTimeZone.forID("Europe/London"));

// Especificar cronología
date = new DateTime(BuddhistChronology.getInstance());
```

## 4.2. Conversión entre fechas de Joda Time y fechas del JDK

```
// de Joda a JDK
DateTime dt = new DateTime();
java.util.Date     date     = dt.toDate();
java.util.Calendar calendar = dt.toCalendar(Locale.US);

// de JDK a Joda
dt = new DateTime(date);
dt = new DateTime(calendar);
```

## 4.3. Campos de fechas y propiedades

Joda Time separa la representación de un instante de tiempo (DateTime) del cálculo de los campos de calendario. Es decir, nosotros tendremos una fecha (representada por un número de milisegundos transcurridos desde "1970-01-01T00:00Z") y, cuando queramos obtener, por ejemplo, el día de la semana, se realizará el cálculo en función de la cronología y zona horaria para obtener el valor correspondiente.

```
DateTime dt = new DateTime();

// Obtener el día de la semana (lunes, martes...)
```

```
int dayOfWeek = dt.getDayOfWeek();

// Obtener el número de minutos transcurridos en el día
int minuteOfDay = dt.getMinuteOfDay();

// Obtener la semana del año
int weekOfYear = dt.getWeekOfWeekyear();
```

La clase DateTimeConstants contiene una serie de constantes enteras con los valores de los días de la semana, los meses, etc.

Además de obtener directamente los valores para los diferentes campos de una fecha, la clase DateTime dispone de métodos para recuperar estos campos como una propiedad (de tipo DateTime.Property). Estas propiedades permiten realizar operaciones adicionales sobre los campos de la fecha. En la documentación oficial de Joda Time se encuentra la lista completa de los campos disponibles para una fecha.

```
DateTime dt = new DateTime();

// Obtener la propiedad correspondiente al día de la semana
Property dayOfWeek = dt.dayOfWeek();

// Obtener el día de la semana como una cadena localizada según la zona horaria ("lunes",
"martes"...)
String sDayOfWeek = dayOfWeek.getAsText();

// Obtener la fecha correspondiente al lunes de la semana actual (las semanas comienzan
en lunes)
DateTime lunes = dayOfWeek.setCopy(DateTimeConstants.MONDAY);
```

Casi todas las clases de Joda Time son "inmutables". Por ejemplo, cuando modifiquemos los campos de una fecha, realmente estaremos obteniendo una nueva fecha con el valor para ese campo modificado. En el ejemplo anterior, el método setCopy obtiene una "copia" de la fecha con el valor modificado, pero no modifica para nada la fecha original.

No siempre será necesario utilizar la clase DateTime.Property para realizar operaciones sobre una fecha. Las operaciones más frecuentes disponen de métodos directos en la clase DateTime:

```
// Sumar dos meses a una fecha
DateTime dosMesesDespues = dt.plusMonths(2);

// Obtener la fecha correspondiente al lunes de la semana actual
DateTime lunes = dt.withDayOfWeek(DateTimeConstants.MONDAY);
```

### 4.4. Formateo de fechas

La clase DateTimeFormatter permite convertir cadenas en fechas y viceversa, utilizando una representación específica de las mismas.

```
// Crear un formatter con una representación específica
DateTimeFormatter fmt = DateTimeFormat.forPattern("dd-MMMM-yyyy");

// Obtener un formatter localizado
DateTimeFormatter americanFmt = fmt.withLocale(Locale.US);

// Obtener una fecha a partir de su representación
DateTime dt = fmt.parseDateTime("25-junio-2010");

// Escribir una fecha con el formato especificado
System.out.println(fmt.print(dt)); // escribe "25-junio-2010"
System.out.println(americanFmt.print(dt)); // escribe "25-June-2010"

// Métodos de acceso directo
System.out.println(dt.toString("dd-MMMM-yyyy")); // escribe "25-junio-2010"
System.out.println(dt.toString("dd-MMMM-yyyy", Locale.US)); // escribe "25-June-2010"
```

También es posible crear formatos utilizando la clase DateTimeFormatterBuilder, aunque para la mayoría de los casos, no será necesario. Esta clase funciona de forma similar a PeriodFormatterBuilder, que se utiliza para crear "formateadores" para los períodos, de la cual se muestra un ejemplo al final del siguiente apartado.

# 5. Trabajar con intervalos y períodos

## 5.1. Intervalos

```
// Crear un intervalo entre el 1 de Enero y la fecha actual
DateTime inicio = new DateTime(2010, 1, 1, 0, 0, 0, 0);
DateTime fin = new DateTime();
Interval interval = new Interval(inicio, fin);

// Recuperar el inicio y fin del intervalo
DateTime i = interval.getStart();
DateTime f = interval.getEnd();

// Comprobar si una fecha determinada está dentro del intervalo
boolean ok = interval.contains(new DateTime("2010-04-13"));

// Conversión a duración o período
Duration duration = interval.toDuration();
Period period = interval.toPeriod();
```

## 5.2. Períodos

```
// ----- Períodos de un único campo -----
// Años desde el 16/03/1976
Years years34 = Years.yearsBetween(new DateTime("1976-03-16"), new DateTime());

// Obtener el número de meses desde un intervalo
Interval interval = new Interval(new DateTime("2010-01-01"), new DateTime("2010-12-01"));
Months months11 = Months.monthsIn(interval);

// Constante predefinida
Days days1 = Days.ONE;

// Especificar un número de días
Days days15 = Days.days(15);

// ----- Períodos de varios campos -----
// 1 año, 6 meses, 2 semanas, 3 días y 12 horas
Period period = new Period(1, 6, 2, 3, 12, 0, 0, 0);

// Años, meses, semanas, días, horas, minutos, segundos y milisegundos desde el
01/01/2000
Period period2 = new Period(new DateTime("2000-01-01"),    new DateTime());

// Años, meses y días desde el 01/01/2000
Period period3 = new Period(new DateTime("2000-01-01"),    new DateTime(),
        PeriodType.yearMonthDay());
```

Como puede verse en los últimos ejemplos, cuando creamos un período sin especificar su tipo, se considera el tipo estándar, que incluye los valores para todos los campos. Si queremos un período que no incluya algunos campos (como las semanas, horas, etc.) debemos especificarlo de manera explícita.

## 5.3. Formateo de períodos

Para terminar vamos a ver un ejemplo de cómo utilizar un PeriodFormatter para convertir una cadena del tipo "HH:mm:ss" a un período que contendrá las horas, minutos y segundos correspondientes, y viceversa. Para crear el Formatter utilizaremos los métodos de la factoría PeriodFormatterBuilder, que permite construir formatos bastante complejos.

```
// Crear el PeriodFormatter
PeriodFormatter durationFormatter = new PeriodFormatterBuilder()
        .minimumPrintedDigits(2) // Número de dígitos que se mostrarán en la salida para
los campos siguientes
        .printZeroAlways() // Indica que deben mostrarse los campos siguientes aunque su
valor sea 0
        .appendHours()
        .appendSeparator(":")
        .appendMinutes()
        .appendSeparator(":")
        .appendSeconds()
        .toFormatter();

// Obtener un período a partir de un String
Period period = durationFormatter.parsePeriod("07:12:38");

// Obtener el String que representa el período
String sPeriod = durationFormatter.print(period);
```

En la construcción del PeriodFormatter, se han utilizado las funciones "minimunPrintedDigits" y "printZeroAlways" al principio. Estos valores se utilizarán para cada campoque se añada después, aunque podrían modificarse para un campo concreto. Por ejemplo, si quisiéramos que para los minutos sólo se imprimiese un dígito y no apareciesen los segundos si su valor fuese cero, podríamos modificar el código anterior como sigue:

```
PeriodFormatter durationFormatter = new PeriodFormatterBuilder()
        .minimumPrintedDigits(2) // Número de dígitos que se mostrarán en la salida para
los campos siguientes
        .printZeroAlways() // Indica que deben mostrarse los campos siguientes aunque su
valor sea 0
        .appendHours()
        .appendSeparator(":")
        .minimumPrintedDigits(1) // Para los minutos queremos mostrar sólamente 1 dígito
        .appendMinutes()
        .appendSeparator(":") // Si los segundos son 0, tampoco se mostrará este
separador
        .minimumPrintedDigits(2) // Pero para los segundos queremos mostrar de nuevo 2
        .printZeroNever() // Si los segundos son 0, no se mostrarán
        .appendSeconds()
        .toFormatter();
```

## 6. Conclusiones

A la vista de lo expuesto en el tutorial, se pueden extraer las siguientes conclusiones sobre la manipulación de fechas en Java:

- Si nuestra aplicación va a utilizar sólamente cálculos comunes con fechas, no hay demasiada diferencia entre utilizar la JDK o Joda Time, por lo cual es más que recomendable utilizar el API estándar.
- En el caso de que tengamos que realizar cálculos o conversiones más complejas, Joda Time nos ofrece un API muy potente capaz de cubrir probablemente todas nuestras necesidades.
- La mayoría de los frameworks utilizados en Java (JSF, Spring...) utilizan las clases del JDK para trabajar con fechas. Si optamos por utilizar Joda Time en nuestras aplicaciones, no deberíamos tener muchos problemas, ya que la conversión entre ambos tipos resulta muy sencilla.
- Joda Time puede hacer que nuestro código sea más sencillo, legible y fácil de entender.
- Si necesitamos trabajar con conceptos como duraciones o períodos, entonces es casi obligatorio el uso de Joda Time, ya que éstos conceptos no están ni mucho menos tan bien soportados en el JDK.