MANEJO DE FECHAS

Desde la versión Java `1.0`, los años en el API empiezan en `1900` y los meses inician con el índice `0`. Por lo tanto, cualquier fecha representada haciendo uso de la clase `java.util.Date` no era legible. Por ejemplo, representaremos la fecha en que Java 8 fue liberado `18 de Marzo del 2014`:

```
Date date = new Date(114, 2, 18);

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

Como resultado de la línea anterior tenemos lo siguiente:

```
Tue Mar 18 00:00:00 AEST 2014
```

En Java `1.1`, la clase `java.util.Calendar` fue agregada y varios métodos de la clase `java.util.Date`fueron deprecados. Hacer uso de la clase `java.util.Calendar` no dio mayores beneficios dado su diseño.

```
Calendar calendar = new GregorianCalendar(2014, 2, 18);

System.out.println(calendar.getTime());
```

A partir de Java 8, la nueva Date API es amigable y esta basada en el proyecto Joda-Time la cual fue adoptada como `JSR-310`.

En el paquete `java.time`, se puede encontrar las nuevas clases del Date API como `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Duration` y `Period`, de las cuales hablaremos a continuación.

LocalDate, LocalTime and LocalDateTime

LocalDate

LocalDate, representa una fecha sin tener en cuenta el tiempo. Haciendo uso del método `of(int year, int month, int dayOfMonth)`, se puede crear un `LocalDate`.

```
LocalDate date = LocalDate.of(1989, 11, 11); //1989-11-11

System.out.println(date.getYear()); //1989

System.out.println(date.getMonth()); //NOVEMBER

System.out.println(date.getDayOfMonth()); //11
```

También, se puede hacer uso del `enum` Month para dar legibilidad al código.

```
LocalDate date = LocalDate.of(1989, Month.NOVEMBER, 11);
```

Finalmente, para capturar el `LocalDate` actual se puede usar el método `now()`:

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

LocalTime

De manera similar, se tiene LocalTime, la cual representa un tiempo determinado. Haciendo uso del método `of()`, esta clase puede crear un `LocalTime` teniendo en cuenta la hora y minuto; hora, minuto y segundo y finalmente hora, minuto, segundo y nanosegundo.

```
LocalTime time = LocalTime.of(5, 30, 45, 35); //05:30:45:35

System.out.println(time.getHour()); //5

System.out.println(time.getMinute()); //30

System.out.println(time.getSecond()); //45

System.out.println(time.getNano()); //35
```

Finalmente, para capturar el `LocalTime` actual se puede usar el método `now()`:

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

LocalDateTime

LocalDateTime, es una clase compuesta, la cual combina las clases anteriormente mencionadas `LocalDate` y `LocalTime`.

En el siguiente fragmento de código se puede apreciar como construir un `LocalDateTime` haciendo uso de todos los campos (año, mes, día, hora, minuto, segundo, nanosegundo):

```
LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35); //1989-11-11T05:30:45.000000035
```

También, se puede crear un objeto `LocalDateTime` basado en los tipos `LocalDate` y `LocalTime`, a continuación se muestra el uso del método `of(LocalDate date, LocalTime time)`:

```
LocalDate date = LocalDate.of(1989, 11, 11);

LocalTime time = LocalTime.of(5, 30, 45, 35);

LocalDateTime dateTime = LocalDateTime.of(date, time);
```

Finalmente, se puede capturar el `LocalDateTime` exacto de la ejecución usando el método `now()`, como se muestra a continuación.

```
LocalDateTime dateTime = LocalDateTime.now();
```

Recordemos que cuando hacemos operaciones sobre las fechas debemos asignar la respuesta a una nueva referencia ya que el objeto original no se modificará puesto que los objetos LocalDate y LocalDataTiem son immutables.

Instant

Instant, representa el número de segundos desde `1 de Enero de 1970`. Es un modelo de fecha y tiempo fácil de interpretar para una máquina.

```
Instant instant = Instant.ofEpochSecond(120);
```

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

El código anterior da como resultado:

```
1970-01-01T00:02:00Z
```

De igual manera que las clases ya mencionadas, `Instant` provee el método `now()`.

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

Duration and Period

Duration

Duration, hace referencia a la diferencia que existe entre dos objetos de tiempo.

En el siguiente ejemplo, la duración se calcula haciendo uso de dos objetos `LocalTime`:

```
LocalTime localTime1 = LocalTime.of(12, 25);

LocalTime localTime2 = LocalTime.of(17, 35);

Duration duration = Duration.between(localTime1, localTime2);
```

Otra opción de calcular la duración entre dos objetos es usando dos objetos `LocalDateTime`:

```
LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14, 13);

LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12, 25);

Duration duration = Duration.between(localDateTime1, localDateTime2);
```

También, se puede crear una duración basada en el método `of(long amount, TemporalUnit unit)`. En el siguiente ejemplo, se muestra como crear un `Duration` de un día.

```
Duration oneDayDuration = Duration.of(1, ChronoUnit.DAYS);
```

Se puede apreciar el uso del enum ChronoUnit, la cual es una implementación de `TemporalUnit` y nos brinda una serie de unidades de períodos de tiempo
como `ERAS`, `MILLENNIA`, `CENTURIES`, `DECADES`, `YEARS`, `MONTHS`, `WEEKS`, etc.

También, se puede crear `Duration` basado en los métodos `ofDays(long days)`, `ofHours(long hours)`, `ofMilis(long milis)`, `ofMinutes(long minutes)`, `ofNanos(long nanos)`, `ofSeconds(long seconds)`.
El ejemplo anterior puede ser reemplazado por la siguiente línea:

```
Duration oneDayDuration = Duration.ofDays(1);
```

Period

Period, hace referencia a la diferencia que existe entre dos fechas.

```
LocalDate localDate1 = LocalDate.of(2016, Month.JULY, 18);

LocalDate localDate2 = LocalDate.of(2016, Month.JULY, 20);

Period period = Period.between(localDate1, localDate2);
```

Se puede crear `Period` basado en el método `of(int years, int months, int days)`. En el siguiente ejemplo, se crea un período de `1 año 2 meses y 3 días`:

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

Del mismo modo que `Duration`, se puede crear `Period` basado en los métodos `ofDays(int days)`, `ofMonths(int months)`, `ofWeeks(int weeks)`, `ofYears(int years)`.

```
Period period = Period.ofYears(1);
```

Manipulación

Manipulando `LocalDate`

Haciendo uso de los métodos `withYear(int year)`, `withMonth(int month)`, `withDayOfMonth(int dayOfMonth)`, `with(TemporalField field, long newValue)` se puede modificar el `LocalDate`.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 25); //2016-07-25

LocalDate date1 = date.withYear(2017); //2017-07-25

LocalDate date2 = date.withMonth(8); //2016-08-25

LocalDate date3 = date.withDayOfMonth(27); //2016-07-27

LocalDate date4 = date.with(ChronoField.MONTH_OF_YEAR, 9); //2016-09-25
```

NOTA: Si en el último ejemplo usamos `ChronoField.HOUR_OF_DAY` la siguiente excepción será lanzada `java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit: HourOfDay`.

Manipulando `LocalTime`

Haciendo uso de los métodos `withHour(int hour)`, `withMinute(int minute)`, `withSecond(int second)`, `withNano(int nanoOfSecond)` se puede modificar el `LocalTime`.

```
LocalTime time = LocalTime.of(14, 30, 35); //14:30:35

LocalTime time1 = time.withHour(20); //20:30:35

LocalTime time2 = time.withMinute(25); //14:25:35

LocalTime time3 = time.withSecond(23); //14:30:23

LocalTime time4 = time.withNano(24); //14:30:35.000000024

LocalTime time5 = time.with(ChronoField.HOUR_OF_DAY, 23); //23:30:35
```

NOTA: Si en el último ejemplo usamos `ChronoField.MONTH_OF_YEAR` la siguiente excepción será lanzada `java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit: MonthOfYear`.

Manipulando `LocalDateTime`

`LocalDateTime` provee los mismo métodos mencionados en las clases `LocalDate` y `LocalTime`.

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 25, 22, 11, 30);

LocalDateTime dateTime1 = dateTime.withYear(2017);
```

```
LocalDateTime dateTime2 = dateTime.withMonth(8);

LocalDateTime dateTime3 = dateTime.withDayOfMonth(27);

LocalDateTime dateTime4 = dateTime.withHour(20);

LocalDateTime dateTime5 = dateTime.withMinute(25);

LocalDateTime dateTime6 = dateTime.withSecond(23);

LocalDateTime dateTime7 = dateTime.withNano(24);

LocalDateTime dateTime8 = dateTime.with(ChronoField.HOUR_OF_DAY, 23);
```

Uso de `TemporalAdjusters`

`LocalDate`, `LocalTime` y `LocalDateTime` proveen los siguientes métodos:

- `with(TemporalField field, long newValue)`
- `with(TemporalAdjuster adjuster)`

El primero de ellos ha sido revisado en los ejemplos anteriores. El segundo brinda una serie de métodos que son útiles para cálculos.

```
import static java.time.temporal.TemporalAdjusters.next;

import static java.time.temporal.TemporalAdjusters.firstDayOfNextMonth;

import static java.time.temporal.TemporalAdjusters.lastDayOfMonth;


LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 25, 22, 11, 30);

LocalDateTime dateTime1 = dateTime.with(TemporalAdjusters.next(DayOfWeek.SUNDAY)); //(1)

LocalDateTime dateTime2 = dateTime.with(TemporalAdjusters.firstDayOfNextMonth()); //(2)

LocalDateTime dateTime3 = dateTime.with(TemporalAdjusters.lastDayOfMonth()); //(3)
```

1. Retorna el próximo Domingo.
2. Retorna el primer día del siguiente mes (Agosto).
3. Retorna el último día del mes (Julio).

Operaciones

Operaciones con `LocalDate`

Realizar operaciones como suma o resta de días, meses, años, etc es muy fácil con la nueva `Date API`. Los siguientes métodos `plus(long amountToAdd, TemporalUnit unit)`, `minus(long amountToSubtract, TemporalUnit unit)` proveen una manera general de realizar estas operaciones.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 18);
```

```
LocalDate datePlusOneDay = date.plus(1, ChronoUnit.DAYS);

LocalDate dateMinusOneDay = date.minus(1, ChronoUnit.DAYS);
```

Asimismo, puede hacerse uso de las siguientes unidades `ChronoUnit.YEARS`, `ChronoUnit.MONTHS`.

NOTA: Si en los ejemplos usamos `ChronoUnit.HOURS` la siguiente excepción será lanzada `java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit: Hours`.

También se puede hacer cálculos basados en un `Period`. En el siguiente ejemplo, se crea un `Period` de 1 día para poder realizar los cálculos.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 18);

LocalDate datePlusOneDay = date.plus(Period.ofDays(1));

LocalDate dateMinusOneDay = date.minus(Period.ofDays(1));
```

Finalmente, haciendo uso de métodos explícitos como `plusDays(long daysToAdd)` y `minusDays(long daysToSubtract)` se puede indicar el valor a incrementar o reducir.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 18);

LocalDate datePlusOneDay = date.plusDays(1);

LocalDate dateMinusOneDay = date.minusDays(1);
```

Operaciones con `LocalTime`

La nueva `Date API` perimite realizar operaciones como suma y resta de horas, minutos, segundos, etc. Al igual que `LocalDate`, los siguientes métodos `plus(long amountToAdd, TemporalUnit unit)`, `minus(long amountToSubtract, TemporalUnit unit)` proveen una manera general de realizar estas operaciones.

```
LocalTime time = LocalTime.of(15, 30);

LocalTime timePlusOneHour = time.plus(1, ChronoUnit.HOURS);

LocalTime timeMinusOneHour = time.minus(1, ChronoUnit.HOURS);
```

Asimismo, puede hacerse uso de las siguientes unidades `ChronoUnit.MINUTES`, `ChronoUnit.SECONDS`, `ChronoUnit.NANOS`.

NOTA: Si en los ejemplos usamos `ChronoUnit.DAYS` la siguiente excepción será lanzada `java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit: Days`.

También se puede hacer cálculos basados en un `Duration`. En el siguiente ejemplo, se crea un `Duration` de 1 hora para poder realizar los cálculos.

```
LocalTime time = LocalTime.of(15, 30);

LocalTime timePlusOneHour = time.plus(Duration.ofHours(1));

LocalTime timeMinusOneHour = time.minus(Duration.ofHours(1));
```

Finalmente, haciendo uso de métodos explícitos como `plusHours(long hoursToAdd)` y `minusHours(long hoursToSubtract)` se puede indicar el valor a incrementar o reducir.

```
LocalTime time = LocalTime.of(15, 30);

LocalTime timePlusOneHour = time.plusHours(1);

LocalTime timeMinusOneHour = time.minusHours(1);
```

Operaciones con `LocalDateTime`

`LocalDateTime`, al ser una clase compuesta por `LocalDate` y `LocalTime` ofrece los mismos métodos para realizar operaciones.

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 28, 14, 30);

LocalDateTime dateTime1 = dateTime.plus(1, ChronoUnit.DAYS).plus(1, ChronoUnit.HOURS);

LocalDateTime dateTime2 = dateTime.minus(1, ChronoUnit.DAYS).minus(1, ChronoUnit.HOURS);
```

En el siguiente ejemplo, se hace uso de `Period` y `Duration`.

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 28, 14, 30);

LocalDateTime dateTime1 = dateTime.plus(Period.ofDays(1)).plus(Duration.ofHours(1));

LocalDateTime dateTime2 = dateTime.minus(Period.ofDays(1)).minus(Duration.ofHours(1));
```

Finalmente, haciendo uso de los métodos `plusX(long xToAdd)` o `minusX(long xToSubtract)`

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 28, 14, 30);

LocalDateTime dateTime1 = dateTime.plusDays(1).plusHours(1);

LocalDateTime dateTime2 = dateTime.minusDays(1).minusHours(1);
```

Además, métodos como `isBefore`, `isAfter`, `isEequal` están disponibles para comparar las siguientes clases `LocalDate`, `LocalTime` y `LocalDateTime`.

```
LocalDate date1 = LocalDate.of(2016, Month.JULY, 28);

LocalDate date2 = LocalDate.of(2016, Month.JULY, 29);


boolean isBefore = date1.isBefore(date2); //true

boolean isAfter = date2.isAfter(date1); //true

boolean isEqual = date1.isEqual(date2); //false
```

Formatos

Cuando se trabaja con fechas, en ocasiones se requiere de un formato personalizado. Java 8 ofrece la clase `java.time.format.DateTimeFormatter` la cual provee algunos formatos.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 25);

String date1 = date.format(DateTimeFormatter.BASIC_ISO_DATE); //20160725

String date2 = date.format(DateTimeFormatter.ISO_DATE); //2016-07-25
```

NOTA: Si en el último ejemplo usamos `DateTimeFormatter.ISO_DATE_TIME` la siguiente excepción será lanzada `java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: HourOfDay`.

También se puede usar el método `ofPattern(String pattern)`, para definir un formato en particular.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 25);

String date1 = date.format(DateTimeFormatter.ofPattern("yyyy/MM/dd")); //2016/07/25
```

Para dar formato a las fechas tenemos algunos comodines o parámetros. Veamos cuales son los más importantes.

- **y**, nos permite acceder al año en formato de cuatro o dos dígitos (2014 o 14).
- **D**, nos permite obtener el número de día del año (225).
- **d,** al contrario del anterior nos devuelve el número del día del mes en cuestión (27).
- **L**, nos ayuda a obtener el mes del año en forma numérica, **M** nos da el mes en texto.
- **H**, nos da la hora.
- **s**, nos da los segundos.
- **m**, nos permite obtener los minutos.
- **a**, nos da el am o pm de la hora.
- **z,** nos permite acceder al nombre de la zona horaria.

Por último, se puede hacer uso de la clase `java.time.format.DateTimeFormatterBuilder`.

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 25, 15, 30);

OffsetDateTime offsetDateTime = dateTime.atOffset(ZoneOffset.ofHours(-5));

DateTimeFormatter formatter = new DateTimeFormatterBuilder()

                                    .appendText(ChronoField.DAY_OF_MONTH)

                                    .appendLiteral(" ")

                                    .appendText(ChronoField.MONTH_OF_YEAR)

                                    .appendLiteral(" ")

                                    .appendText(ChronoField.YEAR)

                                    .appendOffsetId()

                                    .toFormatter();

String dateTime1 = offsetDateTime.format(formatter); //25 July 2016-05:00
```

Utilizar fechas estáticas desde cadenas de texto

Otro uso que podemos darle a las nuevas clases de fecha y hora de **Java** es para poder utilizar fechas que vienen desde una cadena de texto, estás pueden ser creadas por un usuario, venir de un archivo de texto, etc. Pero lo importante es que debemos manipularlas y para ello podemos utilizar todas las herramientas que tenemos a mano.

Veamos en el siguiente código como haciendo uso de lo que hemos visto podemos hacer cambios en una fecha proveniente de una cadena de texto.

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

public class FechaEstatica {

public static void main(String[] args)
{

      String fechaInicial = "1906-12-31";
      LocalDate fechaTomada = LocalDate.parse(fechaInicial);
      System.out.println("Fecha: " + fechaTomada);
      String fechaInicialHora = "1906-12-31T12:05";
      LocalDateTime fechaHoraTomada = LocalDateTime.parse(fechaInicialHora);
      System.out.println("Fecha/Hora: " + fechaHoraTomada);
      DateTimeFormatter df = DateTimeFormatter.ofPattern("dd MMM uuuu");
      System.out.println(fechaTomada + " Con nuevo formato: " +
                                                df.format(fechaTomada));
}
}
```

En el código vemos como creamos una fecha inicial en cadena de texto, luego con un objeto del tipo **LocalDate** utilizamos el método **Parse** para incorporar la cadena de texto a un objeto tipo fecha, luego imprimimos el objeto y vemos que si tomó la fecha adecuadamente, repetimos el mismo procedimiento pero utilizando fecha y hora con **LocalDateTime**.