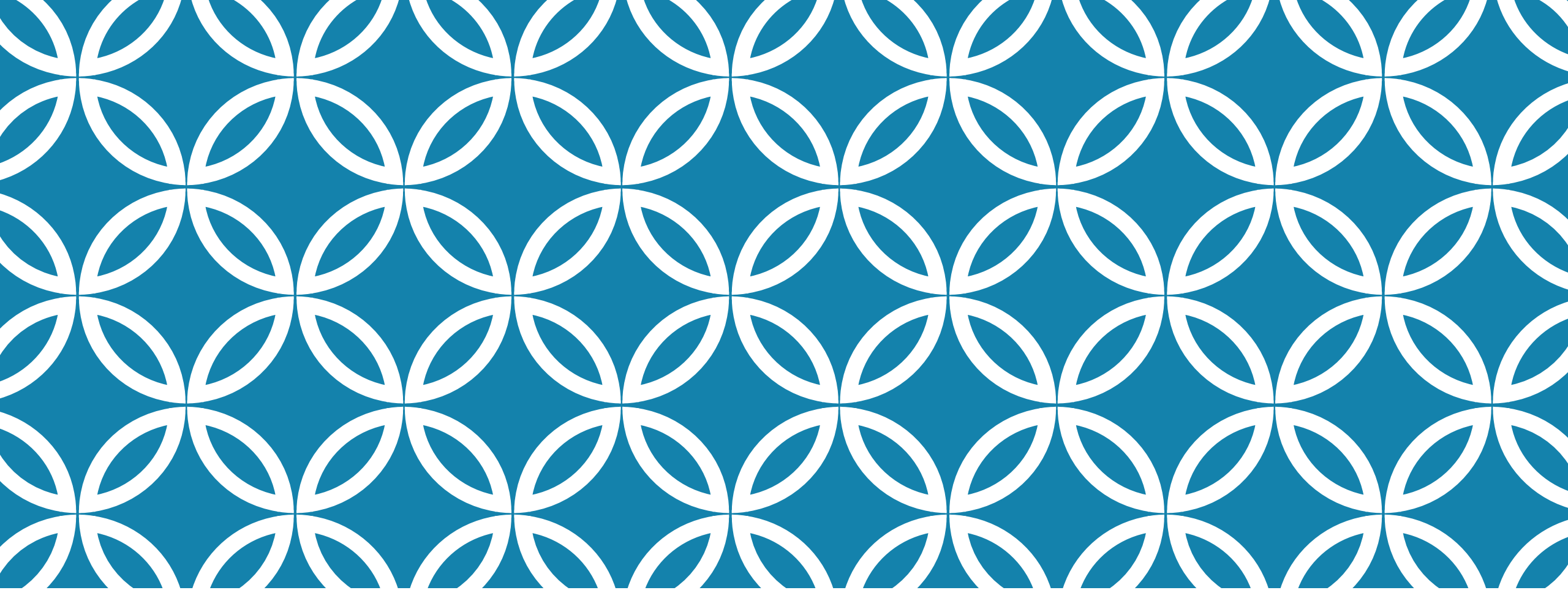

FECHAS CON JAVA.TIME

¿POR QUÉ UN NUEVO PAQUETE DE FECHAS?

El Date es un tipo poco intuitivo, ya que los meses empiezan desde 0, así como no
Contiene métodos amigables para hacer operaciones con fechas.

Uno de los problemas más comunes a los que se enfrentan los desarrolladores es el manejo de fechas, tiempo y zonas horarias, para resolver este problema se solían utilizar api's como Joda Time.

Java 8 incluyó un conjunto de apis que nos ayudan a resolverlo sin incluir dependencias adicionales.

¿POR QUÉ UN NUEVO PAQUETE PARA FECHAS?

Una fecha sin zona horaria en el sistema de calendario ISO-8601, como 2007-12-03. **LocalDate** es un objeto de fecha y hora inmutable que representa una fecha, a menudo vista como año-mes-día. También se puede acceder a otros campos de fecha, como el día del año, el día de la semana y la semana del año.
Por ejemplo, el valor "2 de octubre de 2007" se puede almacenar en un LocalDate.

¿POR QUÉ UN NUEVO PAQUETE DE FECHAS?

- Esta clase no almacena ni representa una hora o zona horaria.
- En cambio, es una descripción de la fecha, tal como se usa para los cumpleaños. No puede representar un instante en la línea de tiempo sin información adicional, como un desplazamiento o zona horaria.
- El sistema de calendario ISO-8601 es el sistema de calendario civil moderno que se usa actualmente en la mayor parte del mundo. Es equivalente al sistema de calendario gregoriano, en el cual las reglas actuales para los años bisiestos se aplican para siempre.
- Para la mayoría de las aplicaciones escritas hoy, las reglas ISO-8601 son totalmente adecuadas.
- Sin embargo, cualquier aplicación que haga uso de fechas históricas y requiera que sean precisas considerará que el enfoque ISO-8601 no es adecuado.

CLASES QUE CONTIENE EL PAQUETE

1. LocalDate
2. LocalTime
3. LocalDateTime
4. Instant
5. Period
6. Duration
7. TemporalUnit
8. DateTimeFormatter

LOCALDATE

LocalDate

Un LocalDate representa una fecha en formato ISO (yyyy-MM-dd) **sin tiempo.** Veamos algunos ejemplos:

```
1  LocalDate date = LocalDate.now();
2  System.out.println(date);
```

El código anterior tendrá la siguiente salida(Entendiendo que el post se escribió el 30-10-2018):

```
1  2018-10-30
```

Como vemos es una fecha **sin tiempo y sin zona horaria.** Veamos algunas otras formas de crear un LocalDate:

```
1  LocalDate date2 = LocalDate.of(2018, 10, 30);
2  LocalDate date3 = LocalDate.parse("2018-10-30");
```

Las 2 expresiones crearán objetos de tipo LocalDate con la fecha del 30-10-2018.

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();
```

OPERACIONES QUE SE PUEDEN REALIZAR CON LOCALDATE

Una vez que sabemos como construir un LocalDate el siguiente paso será conocer el tipo de operaciones que se pueden realizar con el.

- Manipulación de fechas (Sumar o restar días, meses, años, etc):

```
1 LocalDate date = LocalDate.parse("2018-10-30");
2 LocalDate newDate = date.plusDays(10);
3 System.out.println(date);
4 System.out.println(newDate);
```

Salida:

```
1 2018-10-30
2 2018-11-09
```

Como vemos a la fecha inicial se le sumaron 10 días y el mes se actualizó de forma automática, esto nos permite evitar considerar el número de días en un mes, el horario de verano, etc.

```
1 LocalDate date = LocalDate.parse("2018-10-30");
2 LocalDate newDate = date.plusMonths(3);
3 System.out.println(date);
4 System.out.println(newDate);
```

Salida:

```
1 2018-10-30
2 2019-01-30
```

De igual forma podemos hacerlo con los meses y LocalDate resolverá si es necesario cambiar de año.

COMPARACIÓN DE FECHAS

Así como podemos realizar operaciones entre las fechas podemos hacer comparaciones entre ellas, veamos algunos ejemplos:

Valida si una fecha es antes que otra:

```
System.out.println(LocalDate.parse("2018-10-30").isBefore(LocalDate.parse("2018-10-31")));
```

Salida

```
true
```

Valida si un año es bisiesto:

```
System.out.println(LocalDate.parse("2018-10-30").isLeapYear());
```

Salida:

```
false
```

Podemos realizar validaciones muy simples sin necesidad de escribir código complejo.

- Obtener información sobre alguna fecha

El siguiente paso será obtener información sobre alguna fecha en específico veamos algunos ejemplos:

Obtener el día de la semana de mi cumpleaños:

```
System.out.println(LocalDate.parse("2019-08-19").getDayOfWeek());
```

Salida:

```
MONDAY
```

LOCALTIME

LocalTime representa una hora sin la fecha, del mismo modo que con LocalDate podemos crearlo haciendo uso de los métodos now(), parse(..) y of(..), veamos algunos ejemplos:

```
LocalTime time = LocalTime.now();
LocalTime time2 = LocalTime.parse("11:00:59.759");
LocalTime time3 = LocalTime.of(11, 00, 59);
System.out.println(time);
System.out.println(time2);
System.out.println(time3);
```

Salida:

```
11:02:06.198
11:00:59.759
11:00:59
```

Las 3 anteriores son formas válidas de crear un objeto LocalTime. Veamos algunas de las operaciones que podemos realizar.

- Modificar un LocalTime

La primera operación que veremos es como modificar un LocalTime:

```
LocalTime time = LocalTime.parse("11:00:59.759");
LocalTime time2 = time.plusHours(1);
System.out.println(time2);
```

Con el método plusHours crearemos un nuevo LocalTime con la nueva hora calculada.

- Validar un LocalTime

El siguiente punto será hacer validaciones sobre un LocalTime:

```
1 LocalTime time = LocalTime.parse("11:00:59.759");
2 LocalTime time2 = LocalTime.parse("12:00:59.759");
3 System.out.println(time.isBefore(time2));
```

Salida:

```
1 true
```

Usando métodos como isBefore podremos saber si una hora es mayor a otra.

- Extraer información de una hora:

El siguiente paso será extraer solo una parte del objeto LocalTime:

```
1 LocalTime time = LocalTime.parse("11:00:59.759");
2 System.out.println(time.getHour());
```

Salida:

```
1
```

Esto lo utilizaremos en caso de que nuestra aplicación utilice solo la hora para realizar algún proceso.

LOCALDATETIME

La siguiente clase a analizar será **LocalDateTime** la cual representa una combinación entre LocalDate y LocalTime, veamos como crearlo:

```
LocalDateTime dateTime = LocalDateTime.now();
LocalDateTime dateTime1=LocalDateTime.of(2018, 10, 10, 11, 25);
LocalDateTime dateTime2=LocalDateTime.parse("2018-10-10T11:25");

System.out.println(dateTime);
System.out.println(dateTime1);
System.out.println(dateTime2);
```

Salida:

```
2018-10-30T11:46:58.274
2018-10-10T11:25
2018-10-10T11:25
```

En la salida anterior podemos ver como la salida incluye fecha y hora por cada objeto. Veamos algunas de las operaciones que podemos realizar con estos objetos.

- Manipular un LocalDateTime:

De igual forma que con los anteriores podemos realzar manipulaciones sobre el LocalDateTime:

```
LocalDateTime dateTime=LocalDateTime.parse("2018-10-10T11:25");
LocalDateTime newDateTime = dateTime.plusDays(1).plusHours(2);

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

Lo anterior creará un objeto LocalDateTime le agregará 1 día y después 2 horas, recordemos que debemos asignar el resultado a una nueva referencia ya que el objeto original no se modificará sino que se devolverá uno nuevo.

- Manipular un LocalDateTime:

De igual forma que con los anteriores podemos realzar manipulaciones sobre el LocalDateTime:

```
LocalDateTime dateTime=LocalDateTime.parse("2018-10-10T11:25");
LocalDateTime newDateTime = dateTime.plusDays(1).plusHours(2);

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

Lo anterior creará un objeto LocalDateTime le agregará 1 día y después 2 horas, recordemos que debemos asignar el resultado a una nueva referencia ya que el objeto original no se modificará sino que se devolverá uno nuevo.

- Realizar validaciones sobre un LocalDateTime

Como lo vimos en los ejemplos anteriores podemos realizar validaciones sobre el LocalDateTime:

```
LocalDateTime dateTime=LocalDateTime.parse("2018-10-10T11:25");
LocalDateTime dateTime2=LocalDateTime.parse("2019-10-10T11:25");

System.out.println(dateTime.isBefore(dateTime2));
```

Salida:

```
true
```

Se realizará del mismo modo que en los casos anteriores solo que ahora considerará tanto la fecha como la hora.

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

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);
```

DATETIMEFORMATTER

```
/*
 * Clase DateTimeFormatter para dar formato a Fechas y Horas
 * ISO_DATE(2021-11-05)
 * ISO_TIME(11:25:47.624)
 */

System.out.printf("Hora de inicio curso (%s) %n",
        DateTimeFormatter.ISO_TIME.format(horaInicioCurso));


//En caso de querer personalizar el patron de formato
//Instanciar la clase DateTimeFormatter
DateTimeFormatter fPersonalizado = DateTimeFormatter.ofPattern("EEEE, dd MMM, YYYY");
System.out.printf("Hora de inicio curso formato personalizado (%s) %n",
        fPersonalizado.format(fICurso));
```