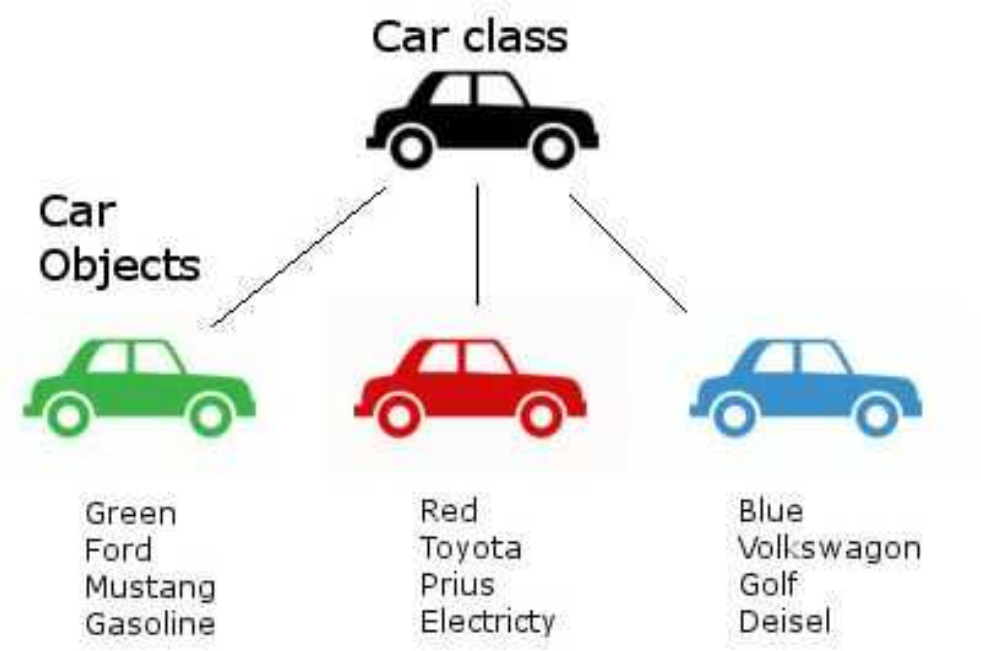
Anexo UD05: Utilización avanzada de clases



1. Wrappers (Envoltorios)

Los wrappers permiten "envolver" datos primitivos en objetos, también se llaman clases contenedoras. La diferencia entre un tipo primitivo y un wrapper es que este último es una clase y por tanto, cuando trabajamos con wrappers estamos trabajando con objetos. Como son objetos debemos tener cuidado en el paso como parámetro en métodos ya que en el wrapper se realiza por referencia.

Una de las principales ventajas del uso de wrappers son la facilidad de conversión entre tipos primitivos y cadenas.

Hay una clase contenedora por cada uno de los tipos primitivos de en Java. Los datos primitivos se escriben en minúsculas y los wrappers se escriben con la primera letra en mayúsculas.

Tipo primitivo	Wrapper asociado
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

Cada clase wrapper tiene dos constructores, uno se le pasa por parámetro el dato de tipo primitivo y otro se le pasa un `String`.

Para wrapper `Integer`:

```
1 Integer(int)
2 Integer(String)
```

Ejemplo:

```
1 Integer i1 = new Integer(42);
2 Integer i2 = new Integer ("42");
3 Float f1 = new Float(3.14f);
4 Float f2 = new Float ("3.14f");
```

Antiguamente, una vez asignado un valor a un objeto o wrapper `Integer`, este no podía cambiarse. Actualmente e internamente realizar un apoyo en variables y wrapers internos para poder variar el valor de un wrapper.

Ejemplo:

```
Integer y = new Integer(567);       //Crea el objeto
y++;                                //Lo desenvuelve, incrementa y lo vuelve a envolver
System.out.println("Valor: " + y); //Imprime el valor del Objeto y
```

Los wrapper disponen de una serie de métodos que permiten realizar funciones de conversión de datos. Por ejemplo, el wrapper `Integer` dispone de los siguientes métodos:

Método	Descripción
Integer(int) Integer(String)	Constructores
byteValue() shortValue() intValue() longValue() doubleValue() floatValue()	Funciones de conversión con datos primitivos
Integer decode(String) Integer parseInt(String) Integer parseInt(String, int) Integer valueOf(String) String toString()	Conversión a String
String toBinaryString(int) String toHexString(int) String toOctalString(int)	Conversión a otros sistemas de numeración
MAX_VALUE, MIN_VALUE, TYPE	Constantes

1.1. Métodos `valueOf()`.

El método `valueOf()` permite crear objetos wrapper y se le pasa un parámetro `String` y opcionalmente otro parámetro que indica la base en la que será representado el primer parámetro.

Ejemplo:

```
// Convierte el 101011 (base 2) a 43 y le asigna el valor al objeto Integer i1
Integer i3 = Integer.valueOf("101011", 2);
System.out.println(i3);

// Asigna 3.14 al objeto Float f2
Float f3 = Float.valueOf("3.14f");
System.out.println(f3);
```

Métodos `xxxValue()`.

Los métodos `xxxValue()` permiten convertir un wrapper en un dato de tipo primitivo y no necesitan argumentos.

Ejemplo:

```
Integer i4 = 120; // Crea un nuevo objeto wrapper
byte b = i4.byteValue(); // Convierte el valor de i2 a un primitivo byte
short s1 = i4.shortValue(); // Otro de los métodos de Integer
double d = i4.doubleValue(); // Otro de los métodos xxxValue de Integer
System.out.println(s1); // Muestra 120 como resultado

Float f4 = 3.14f; // Crea un nuevo objeto wrapper
short s2 = f4.shortValue(); // Convierte el valor de f2 en un primitivo short
System.out.println(s2); // El resultado es 3 (truncado, no redondeado)
```

1.2. Métodos `parseXxxx()`.

Los métodos `parseXxxx()` permiten convertir un wrapper en un dato de tipo primitivo y le pasamos como parámetro el `String` con el valor que deseamos convertir y opcionalmente la base a la que convertiremos el valor (2, 8, 10 o 16).

Ejemplo:

```
double d4 = Double.parseDouble("3.14"); // Convierte un String a primitivo
System.out.println("d4 = " + d4);   // El resultado será d4 = 3.14
long l2 = Long.parseLong("101010", 2);  // un String binario a primitivo
System.out.println("l2 = " + l2);   // El resultado es L2 42
```

1.3. Métodos `toString()`.

El método `toString()` permite retornar un `String` con el valor primitivo que se encuentra en el objeto contenedor. Se le pasa un parámetro que es el wrapper y opcionalmente para `Integer` y `Long` un parámetro con la base a la que convertiremos el valor (2, 8, 10 o 16).

Ejemplo:

```
Double d1 = new Double("3.14");
System.out.println("d1 = " + d1.toString() ); // El resultado es d = 3.14
String d2 = Double.toString(3.14); // d2 = "3.14"
System.out.println("d2 = " + d2); // El resultado es d = 3.14
String s3 = "hex = " + Long.toString(254, 16); // s = "hex = fe"
System.out.println("s3 = " + s3); // El resultado es d = 3.14
```

1.4. Métodos `toXxxxxString()` (Binario, Hexadecimal y Octal).

Los métodos `toXxxxxString()` permiten a las clases contenedoras `Integer` y `Long` convertir números en base 10 a otras bases, retornando un `String` con el valor primitivo que se encuentra en el objeto contenedor.

Ejemplo:

```
String s4 = Integer.toHexString(254); // Convierte 254 a hex
System.out.println("254 es " + s4); // Resultado: "254 es fe"
String s5 = Long.toOctalString(254); // Convierte 254 a octal
System.out.println("254(oct) = " + s5); // Resultado: "254(oct) = 376"
```

Para resumir, los métodos esenciales para las conversiones son:

- `primitive xxxValue()` – Para convertir de Wrapper a primitive
- `primitive parseXxx(String)` – Para convertir un String en primitive
- **Wrapper** `valueOf(String)` – Para convertir String en Wrapper

2. Clase `Date`

La clase Date es una utilidad contenida en el paquete `java.util` y permiten trabajar con fechas y horas. La fechas y hora se almacenan en un entero de tipo `Long` que almacena los milisegundos transcurridos desde el 1 de Enero de de 1970 que se obtienen con `getTime()`. (Importamos `java.util.Date`).

Ejemplo:

```
Date fecha = new Date(2021, 8, 19);
System.out.println(fecha);           //Mon Sep 19 00:00:00 CEST 3921
System.out.println(fecha.getTime()); //61590146400000
```

2.1. Clase `GregorianCalendar`.

Para utilizar fechas y horas se utiliza la clase `GregorianCalendar` que dispone de variable enteras como: `DAY_OF_WEEK`, `DAY_OF_MONTH`, `YEAR`, `MONTH`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`, `WEEK_OF_MONTH`, `WEEK_OF_YEAR`, ... (Importamos Clase `java.util.Calendar` y `java.util.GregorianCalendar`)

Ejemplo 1:

```
Calendar calendar = new GregorianCalendar(2021, 8, 19);
System.out.println(calendar.getTime()); //Sun Sep 19 00:00:00 CEST 2021
```

Ejemplo 2:

```
Date d = new Date();
GregorianCalendar c = new GregorianCalendar();
System.out.println("Fecha: "+d);  //Fecha: Thu Aug 19 20:06:14 CEST 2021
System.out.println("Info: "+c); //Info:
java.util.GregorianCalendar[time=1629396374723,

//areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInf
o

//[id="Europe/Madrid",offset=3600000,dstSavings=3600000,useDaylight=true,transitions
=163,

 //lastRule=java.util.SimpleTimeZone[id=Europe/Madrid,offset=3600000,dstSavings=3600
000,

 //useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=
1,

 //startTime=3600000,startTimeMode=2,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,

 //endTime=3600000,endTimeMode=2]],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1,

 //YEAR=2021,MONTH=7,WEEK_OF_YEAR=33,WEEK_OF_MONTH=3,DAY_OF_MONTH=19,DAY_OF_YEAR=231
,
```

```
12
     //DAY_OF_WEEK=5,DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=8,HOUR_OF_DAY=20,MINUTE=6,SECON
    D=14,
13
     //MILLISECOND=723,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
14  c.setTime(d);
15  System.out.print(c.get(Calendar.DAY_OF_MONTH));
16  System.out.print("/");
17  System.out.print(c.get(Calendar.MONTH)+1);
18  System.out.print("/");
19  System.out.println(c.get(Calendar.YEAR)+1); //19/8/2022
```

2.2. Paquete `java.time`.

El paquete `java.time` dispone de las clases `LocalDate`, `LocalTime`, `LocalDateTime`, `Duration` y `Period` para trabajar con fechas y horas.

Estas clases no tienen constructores públicos, y por tanto, no se puede usar `new` para crear objetos de estas clases. Necesitas usar sus métodos `static` para instanciarlas.

No es válido llamar directamente al constructor usando `new`, ya que no tienen un constructor público.

Ejemplo:

```
1  LocalDate d = new LocalDate(); //NO compila
```

2.2.1. `LocalDate`

`LocalDate` representa una fecha determinada. Haciendo uso del método `of()`, esta clase puede crear un `LocalDate` teniendo en cuenta el año, mes y día. Finalmente, para capturar el `LocalDate` actual se puede usar el método `now()`:

Ejemplo:

```
1  LocalDate date = LocalDate.of(1989, 11, 11); //1989-11-11
2  System.out.println(date.getYear()); //1989
3  System.out.println(date.getMonth()); //NOVEMBER
4  System.out.println(date.getDayOfMonth()); //11
5  date = LocalDate.now();
6  System.out.println(date); //2021-08-19
```

2.2.2. `LocalTime`

`LocalTime`, representa un tiempo determinado. Haciendo uso del método `of()`, esta clase puede crear un `LocalTime` teniendo en cuenta la hora, minuto, segundo y nanosegundo. Finalmente, para capturar el `LocalTime` actual se puede usar el método `now()`.

```
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
time = LocalTime.now();
System.out.println(time); //20:13:53.118044
```

2.2.3. `LocalDateTime`

`LocalDateTime`, es una clase compuesta, la cual combina las clases anteriormente mencionadas `LocalDate` y `LocalTime`. Podemos construir un `LocalDateTime` haciendo uso de todos los campos (año, mes, día, hora, minuto, segundo, nanosegundo).

Ejemplo:

```
LocalDateTime dateTime = LocalDateTime.of (1989, 11, 11, 5, 30, 45, 35);
```

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

Ejemplo:

```
LocalDate date = LocalDate.of(1989, 11, 11);
LocalTime time = LocalTime.of(5, 30, 45, 35);
LocalDateTime dateTime = LocalDateTime.of(date, time);
LocalDateTime dateTime = LocalDateTime.now();
```

2.2.4. `Duration`

`Duration`, hace referencia a la diferencia que existe entre dos objetos de tiempo. La duración denota la cantidad de tiempo en horas, minutos y segundos.

Ejemplo:

```
LocalTime localTime1 = LocalTime.of(12, 25);
LocalTime localTime2 = LocalTime.of(17, 35);
Duration duration1 = Duration.between(localTime1, localTime2);
System.out.println(duration1); //PT5H10M
System.out.println(duration1.toDays()); //0

LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14, 13);
LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12, 25);
Duration duration2 = Duration.between(localDateTime1, localDateTime2);
System.out.println(duration2); //PT46H12M
System.out.println(duration2.toDays()); //1
```

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)`.

Ejemplo:

```
Duration duracion3 = Duration.ofDays(1);
System.out.println(duracion3); //PT24H
System.out.println(duracion3.toDays()); //1
```

2.2.5. `Period`

`Period`, hace referencia a la diferencia que existe entre dos fechas. Esta clase denota la cantidad de tiempo en años, meses y días.

```
LocalDate localDate1 = LocalDate.of(2016, 7, 18);
LocalDate localDate2 = LocalDate.of(2016, 7, 20);
Period periodo1 = Period.between(localDate1, localDate2);
System.out.println(periodo1); //P2D
```

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 periodo2 = Period.of(1, 2, 3);
System.out.println(periodo2); //P1Y2M3D
```

Se puede crear `Period` basado en los métodos `ofDays(int days)`, `ofMonths(int months)`, `ofWeeks(int weeks)`, `ofYears(int years)`.

Ejemplo:

```
Period periodo3 = Period.ofYears(1);
System.out.println(periodo3); //P1Y
```

2.3. `ChronoUnit`

Permite devolver el tiempo transcurrido entre dos fechas en diferentes formatos (`DAYS`, `MONTHS`, `YEARS`, `HOURS`, `MINUTES`, `SECONDS`, ...). Debemos importar la clase `time.temporal.ChronoUnit`;

Ejemplo:

```
LocalDate fechaInicio = LocalDate.of(2016, 7, 18);
LocalDate fechaFin = LocalDate.of(2016, 7, 20);
// Calculamos el tiempo transcurrido entre las dos fechas
// con la clase ChronoUnit y la unidad temporal en la que
// queremos que nos lo devuelva, en este caso DAYS.
long tiempo = ChronoUnit.DAYS.between(fechaInicio, fechaFin);
System.out.println(tiempo); //2
```

2.4. Introducir fecha como Cadena.

Podemos introducir la fecha como una cadena con el formato que deseemos y posteriormente convertir a fecha con la sentencia `parse`. Debemos importar las clases `time` y `time.format`.

Ejemplo:

```
DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
String fechaCadena = "16/08/2016";
LocalDate mifecha = LocalDate.parse(fechaCadena, formato);
System.out.println(formato.format(mifecha)); //16/08/2016
```

2.5. Manipulación

1. 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`.

   Ejemplo:

   ```
   LocalDate date = LocalDate.of(2016, 7, 25);
   LocalDate date1 = date.withYear(2017);
   LocalDate date2 = date.withMonth(8);
   LocalDate date3 = date.withDayOfMonth(27);
   System.out.println(date);  //2016-07-25
   System.out.println(date1); //2017-07-25
   System.out.println(date2); //2016-08-25
   System.out.println(date3); //2016-07-27
   ```

2. Manipulando `LocalTime`

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

   Ejemplo:

   ```
   LocalTime time = LocalTime.of(14, 30, 35);
   LocalTime time1 = time.withHour(20);
   LocalTime time2 = time.withMinute(25);
   LocalTime time3 = time.withSecond(23);
   LocalTime time4 = time.withNano(24);
   System.out.println(time);  //14:30:35
   System.out.println(time1); //20:30:35
   System.out.println(time2); //14:25:35
   System.out.println(time3); //14:30:23
   System.out.println(time4); //14:30:35.000000024
   ```

3. Manipulando `LocalDateTime`

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

   Ejemplo:

   ```
   LocalDateTime dateTime = LocalDateTime.of(2016, 7, 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);
   ```

```
System.out.println(dateTime);  //2016-07-25T22:11:30
System.out.println(dateTime1); //2017-07-25T22:11:30
System.out.println(dateTime2); //2016-08-25T22:11:30
System.out.println(dateTime3); //2016-07-27T22:11:30
System.out.println(dateTime4); //2016-07-25T20:11:30
System.out.println(dateTime5); //2016-07-25T22:25:30
System.out.println(dateTime6); //2016-07-25T22:11:23
System.out.println(dateTime7); //2016-07-25T22:11:30.000000024
```

2.6. Operaciones

1. 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. (Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.YEARS`, `ChronoUnit.MONTHS`, `ChronoUnit.DAYS`).

   Ejemplo:

   ```
   LocalDate date = LocalDate.of(2016, 7, 18);
   LocalDate datePlusOneDay = date.plus(1, ChronoUnit.DAYS);
   LocalDate dateMinusOneDay = date.minus(1, ChronoUnit.DAYS);
   System.out.println(date);            // 2016-07-18
   System.out.println(datePlusOneDay);  // 2016-07-19
   System.out.println(dateMinusOneDay); // 2016-07-17
   ```

   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.

   Ejemplo:

   ```
   LocalDate date = LocalDate.of(2016, 7, 18);
   LocalDate datePlusOneDay = date.plus(Period.ofDays(1));
   LocalDate dateMinusOneDay = date.minus(Period.ofDays(1));
   System.out.println(date);            // 2016-07-18
   System.out.println(datePlusOneDay);  // 2016-07-19
   System.out.println(dateMinusOneDay); // 2016-07-17
   ```

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

   Ejemplo:

   ```
   LocalDate date = LocalDate.of(2016, 7, 18);
   LocalDate datePlusOneDay = date.plusDays(1);
   LocalDate dateMinusOneDay = date.minusDays(1);
   System.out.println(date);            // 2016-07-18
   System.out.println(datePlusOneDay);  // 2016-07-19
   System.out.println(dateMinusOneDay); // 2016-07-17
   ```

2. 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.

(Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.HOURS`, `ChronoUnit.MINUTES`, `ChronoUnit.SECONDS`, `ChronoUnit.NANOS`).

Ejemplo:

```
LocalTime time = LocalTime.of(15, 30);
LocalTime timePlusOneHour = time.plus(1, ChronoUnit.HOURS);
LocalTime timeMinusOneHour = time.minus(1, ChronoUnit.HOURS);
System.out.println(time);               // 15:30
System.out.println(timePlusOneHour);    // 16:30
System.out.println(timeMinusOneHour);   // 14:30
```

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));
System.out.println(time);               // 15:30
System.out.println(timePlusOneHour);    // 16:30
System.out.println(timeMinusOneHour);   // 14:30
```

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

Ejemplo:

```
LocalTime time = LocalTime.of(15, 30);
LocalTime timePlusOneHour = time.plusHours(1);
LocalTime timeMinusOneHour = time.minusHours(1);
System.out.println(time);               // 15:30
System.out.println(timePlusOneHour);    // 16:30
System.out.println(timeMinusOneHour);   // 14:30
```

3. Operaciones con `LocalDateTime`

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

(Debemos importar la clase `java.time.temporal.ChronoUnit` para poder utilizar las unidades: `ChronoUnit.YEARS`, `ChronoUnit.MONTHS`, `ChronoUnit.DAYS`, `ChronoUnit.HOURS`, `ChronoUnit.MINUTES`, `ChronoUnit.SECONDS`, `ChronoUnit.NANOS`).

Ejemplo:

```
LocalDateTime dateTime = LocalDateTime.of(2016, 7, 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);
System.out.println(dateTime);  // 2016-07-28T14:30
System.out.println(dateTime1); // 2016-07-29T15:30
System.out.println(dateTime2); // 2016-07-27T13:30
```

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

```
LocalDateTime dateTime = LocalDateTime.of(2016, 7, 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));
System.out.println(dateTime);  // 2016-07-28T14:30
System.out.println(dateTime1); // 2016-07-29T15:30
System.out.println(dateTime2); // 2016-07-27T13:30
```

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

```
LocalDateTime dateTime = LocalDateTime.of(2016, 7, 28, 14, 30);
LocalDateTime dateTime1 = dateTime.plusDays(1).plusHours(1);
LocalDateTime dateTime2 = dateTime.minusDays(1).minusHours(1);
System.out.println(dateTime);  // 2016-07-28T14:30
System.out.println(dateTime1); // 2016-07-29T15:30
System.out.println(dateTime2); // 2016-07-27T13:30
```

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

Ejemplo:

```
LocalDate date1 = LocalDate.of(2016, 7, 28);
LocalDate date2 = LocalDate.of(2016, 7, 29);
boolean isBefore = date1.isBefore(date2); //true
boolean isAfter = date2.isAfter(date1); //true
boolean isEqual = date1.isEqual(date2); //false
```

2.7. Formatos

Cuando se trabaja con fechas, en ocasiones se requiere de un formato personalizado. Podemos usar el método `ofPattern(String pattern)`, para definir un formato en particular.

Para utilizar `DateTimeFormatter.ofPattern` debemos importar la clase con `import java.time.format.DateTimeFormatter;`

Ejemplo:

```
LocalDate mifecha = LocalDate.of(2016, 7, 25);
String fechaTexto=mifecha.format(DateTimeFormatter.ofPattern("eeee',' dd 'de' MMMM 'del' yyyy"));
System.out.println("La fecha es: "+fechaTexto); // La fecha es: lunes, 25 de julio del 2016
```

El patrón del formato se realiza en función a la siguiente tabla de símbolos:

Símbolo	Descripción	Salida
y	Año	2004; 04
D	Día del Año	189
M	Mes del Año	7; 07; Jul; July; J
d	Día del Mes	10
w	Semana del Año	27
EEE	Día de la Semana	Tue; Tuesday; T
F	Semana del Mes	3
a	AM/PM	PM
K	Hora AM/PM (0-11)	0
H	Hora del día (0-23)	0
m	Minutos de la hora	30
s	Segundos del minuto	55
n	Nanosegundos del Segundo	987654321
''	Texto	'Día de la semana'

2.7.1. Día de la Semana.

La función `getDayOfWeek()` devuelve un elemento del tipo `DayOfWeek` que corresponde el día de la semana de una fecha. Debemos importar la clase `java.time.DayOfWeek`.

Por ejemplo, el lunes será `DayOfWeek.MONDAY`.

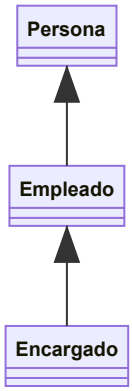
Ejemplo:

```
LocalDate lafecha = LocalDate.of(2016, 7, 25);
if (lafecha.getDayOfWeek().equals(DayOfWeek.SATURDAY)) {
    System.out.println("La fecha es Sábado");
} else {
    System.out.println("La fecha NO es Sábado");
}
//La fecha NO es Sábado
```

3. Conversión entre objetos (Casting).

La esencia de Casting permite convertir un dato de tipo primitivo en otro generalmente de más precisión.

Entre objetos es posible realizar el casting. Tenemos una clase persona con una subclase empleado y este a su vez una subclase encargado.



Si creamos una instancia de tipo persona y le asignamos un objeto de tipo empleado o encargado, al ser una subclase no existe ningún tipo de problema, ya que todo encargado o empleado es persona.

Por otro lado, si intentamos asignar valores a los atributos específicos de empleado o encargado nos encontramos con una pérdida de precisión puesto que no se pueden ejecutar todos los métodos de los que dispone un objeto de tipo empleado o encargado, ya que persona contiene menos métodos que la clase empleado o encargado. En este caso es necesario hacer un casting, sino el compilador dará error.

Ejemplo:

```
// Clase Personal que solo dispone de nombre
class Persona {

    String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    @Override
    public String toString() {
        return "Nombre: " + nombre + "\n";
    }
}
```

```java
// Clase Empleado que hereda de Persona y añade atributo sueldoBase
class Empleado extends Persona {

    double sueldoBase;

    public Empleado(String nombre, double sueldoBase) {
        super(nombre);
        this.sueldoBase = sueldoBase;
    }

    public double getSueldo() {
        return sueldoBase;
    }

    public void setSueldoBase(double sueldoBase) {
        this.sueldoBase = sueldoBase;
    }

    @Override
    public String toString() {
        return super.toString() + "Sueldo Base: " + sueldoBase + "\n";
    }
}
```

```java
// Clase Encargado que hereda de Empleado y añade atributo seccion
class Encargado extends Empleado {

    String seccion;

    public Encargado(String nombre, double sueldoBase, String seccion) {
        super(nombre, sueldoBase);
        this.seccion = seccion;
    }

    public String getSeccion() {
        return seccion;
    }

    public void setSeccion(String seccion) {
        this.seccion = seccion;
    }

    @Override
    public String toString() {
        return super.toString() + "Sección:" + seccion + "\n";
    }
}
```

```java
public class Casting {
    //Refundición de Objetos
    public static void main(String[] args) {
        // Casting Implicito
        Persona encargadoCarniceria = new Encargado("Rosa Ramos", 0, null);
        // Casting Explicito
        Encargado miEncargado = (Encargado) encargadoCarniceria;
        miEncargado.setSueldoBase(1200);
        miEncargado.setSeccion("Carniceria");
        // Muestra los datos de la Persona que es Encargado
        System.out.println(encargadoCarniceria.toString());
    }
}
```

Las reglas a la hora de realizar casting es que:

- cuando se utiliza una clase más específicas (más abajo en la jerarquía) no hace falta casting. Es lo que llamamos **casting implícito**.
- cuando se utiliza una clase menos específica (más arriba en la jerarquía) hay que hacer un **casting explícito**.

4. Acceso a métodos de la superclase.

Para acceder a los métodos de la superclase se utiliza la sentencia `super`. La sentencia `this` permite acceder a los campos y métodos de la clase y la sentencia super permite acceder a los campos y métodos de la superclase. Ejemplo

```
public class Persona {

    private String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```
public class Deportista extends Persona{
    public Deportista(String nombre) {
        super(nombre);
    }
 }
```

Podemos mostrar el nombre de la clase y el nombre de la clase de la que hereda con `getClass()` y `getSuperclass()`. Ejemplo:

```
public class EjemploGetClass {

    public static void main(String[] args) {
        Deportista deportistaNatacion = new Deportista("Rosa Ramos");

        // Muestra los datos de la Persona que es Deportista
        System.out.println(deportistaNatacion instanceof Vehiculo); //false
        System.out.println(deportistaNatacion.getClass()); //class Deportista
        System.out.println(deportistaNatacion.getClass().getSuperclass()); //class
Persona
    }
}
```

5. Clases Anidadas, Clases Internas (Inner Class).

Una clase anidada es una clase que es miembro de otra clase. La clase anidada al ser miembro de la clase externa tienen acceso a todos sus métodos y atributos.

Permiten:

- acceder a los campos privados de la otra clase.
- ocultar la clase interna de las otras clases del paquete.
- Permite gestionar eventos.

```
class Externa{
    ...
    class Interna{
        ...
    }
    ...
}
```

Para instanciar una clase interna se utilizará la sentencia:

```
Externa.Interna objetoInterno = objetoExterno.new Interna();
```

Ejemplo:

```
class Pc {

    double precio;

    public String toString() {
        return "El precio del PC es " + this.precio;
    }

    class Monitor {

        String marca;

        public String toString() {
            return "El monitor es de la marca " + this.marca;
        }
    }

    class Cpu {

        String marca;

        public String toString() {
            return "La CPU es de la marca " + this.marca;
        }
    }
}

public class ClaseInternaHardware {
```

```
29
30      public static void main(String[] args) {
31          Pc miPc = new Pc();
32          Pc.Monitor miMonitor = miPc.new Monitor();
33          Pc.Cpu miCpu = miPc.new Cpu();
34          miPc.precio = 1250.75;
35          miMonitor.marca = "Asus";
36          miCpu.marca = "Acer";
37          System.out.println(miPc); //El precio del PC es 1250.75
38          System.out.println(miMonitor); //El monitor es de la marca Asus
39          System.out.println(miCpu); //La CPU es de la marca Acer
40      }
41  }
```

6. Interfaces Java

Supongamos una situación en la que nos interesa dejar constancia de que ciertas clases deben implementar una funcionalidad teórica determinada, diferente en cada clase afectada. Estamos hablando, pues, de la definición de un método teórico que algunas clases deberán implementar.

Un ejemplo real puede ser el método `calculoImporteJubilacio()` aplicable, de manera diferente, a muchas tipologías de trabajadores y, por tanto, podríamos pensar en diseñar una clase `Trabajador` en que uno de sus métodos fuera `calculpImporteJubilacion()`. Esta solución es válida si estamos diseñando una jerarquía de clases a partir de la clase `Trabajador` de la que cuelguen las clases correspondientes a las diferentes tipologías de trabajadores (metalúrgicos, hostelería, informáticos, profesores ...). Además, disponemos del concepto de clase abstracta que cada subclase implemente obligatoriamente el método `calculoImporteJubilacion()`.

Pero, ¿y si resulta que ya tenemos las clases `Profesor`, `Informatico`, `Hostelero` en otras jerarquías de clases? La solución consiste en hacer que estas clases derivaran de la clase `Trabajador`, sin abandonar la derivación que pudieran tener, sería factible en lenguajes orientados a objetos que soportaran la herencia múltiple, pero esto no es factible en el lenguaje Java.

Para superar esta limitación, Java proporciona las interfaces.

> Una interfaz es una **maqueta** contenedora de una lista de métodos abstractos y datos miembro (de tipos primitivos o de clases). Los atributos, si existen, son implícitamente consideradas `static` y `final`. Los métodos, si existen, son implícitamente considerados `public`.

Para entender en qué nos pueden ayudar las interface, necesitamos saber:

- Una interfaz puede ser implementada por múltiples clases, de manera similar a como una clase puede ser superclase de múltiples clases.
- Las clases que implementan una interfaz están obligadas a sobrescribir todos los métodos definidos en la interfaz. Si la definición de alguno de los métodos a sobrescribir coincide con la definición de algún método heredado, este desaparece de la clase.
- Una clase puede implementar múltiples interfaces, a diferencia de la derivación, que sólo se permite una única clase base.
- Una interfaz introduce un nuevo tipo de dato, por la que nunca habrá ninguna instancia, pero sí objetos usuarios de la interfaz (objetos de las clases que implementan la interfaz). Todas las clases que implementan una interfaz son compatibles con el tipo introducido por la interfaz.
- Una interfaz no proporciona ninguna funcionalidad a un objeto (ya que la clase que implementa la interfaz es la que debe definir la funcionalidad de todos los métodos), pero en cambio proporciona la posibilidad de formar parte de la funcionalidad de otros objetos (pasándola por parámetro en métodos de otras clases).
- La existencia de las interfaces posibilita la existencia de una jerarquía de tipo (que no debe confundirse con la jerarquía de clases) que permite la herencia múltiple.
- Una interfaz no se puede instanciar, pero sí se puede hacer referencia.

  > Así, si `I` es una interfaz y `C` es una clase que implementa la interfaz, se pueden declarar referencias al tipo `I` que apunten objetos de `C`:

```
I obj = new C (<parámetros>);
```

- Las interfaces pueden heredar de otras interfaces y, a diferencia de la derivación de clases, pueden heredar de más de una interfaz.

Así, si diseñamos la interfaz `Trabajador`, podemos hacer que las clases ya existentes (`Profesor`, `Informatico`, `Hostelero` ...) la implementen y, por tanto, los objetos de estas clases, además de ser objetos de las superclases respectivas, pasan a ser considerados objetos usuarios del tipo `Trabajador`. Con esta actuación nos veremos obligados a implementar el método `calculoImporteJubilacion()` a todas las clases que implementen la interfaz.

Alguien no experimentado en la gestión de interfaces puede pensar: ¿por qué tanto revuelo con las interfaces si hubiéramos podido diseñar directamente un método llamado `calculoImporteJubilacion()` en las clases afectadas sin necesidad de definir ninguna interfaz?

La respuesta radica en el hecho de que la declaración de la interfaz lleva implícita la declaración del tipo `Trabajador` y, por tanto, podremos utilizar los objetos de todas las clases que implementen la interfaz en cualquier método de cualquier clase que tenga algún argumento referencia al tipo `Trabajador` como, por ejemplo, en un hipotético método de una hipotética clase llamada Hacienda:

```
public void enviarBorradorIRPF(Trabajador t) {...}
```

Por el hecho de existir la interfaz `Trabajador`, todos los objetos de las clases que la implementan (`Profesor`, `Informatico`, `Hostelero` ...) se pueden pasar como parámetro en las llamadas al método `enviarBorradorIRPF(Trabajador t)`.

La sintaxis para declarar una interfaz es:

```
[public] interface <NombreInterfaz> [extends <Nombreinterfaz1>, <Nombreinterfaz2>...]
{
    <CuerpoInterfaz>
}
```

Las interfaces también se pueden asignar a un paquete. La inexistencia del modificador de acceso público hace que la interfaz sea accesible a nivel del paquete.

Para los nombres de las interfaces, se aconseja seguir el mismo criterio que para los nombres de las clases.

> En la documentación de Java, las interfaces se identifican rápidamente entre las clases porque están en cursiva.

El cuerpo de la interfaz es la lista de métodos y/o constantes que contiene la interfaz. Para las constantes no hay que indicar que son `static` y `final` y para los métodos no hay que indicar que son `public`. Estas características se asignan implícitamente.

La sintaxis para declarar una clase que implemente una o más interfaces es:

```
[final] [public] class <NombreClase> [extends <NombreClaseBase>] implements
<NombreInterfaz1>, <NomInterfaz2>... {
    <CuerpoDeLaClase>
}
```

Los métodos de las interfaces a implementar en la clase deben ser obligatoriamente de acceso público.

Por último, comentar que, como por definición todos los datos miembro que se definen en una interfaz son `static` y `final`, y dado que las interfaz no se pueden instanciar, también resultan una buena herramienta para implantar grupos de constantes.

Así, por ejemplo:

```
public interface DiasSemana {
    int LUNES = 1, MARTES=2, MIERCOLES=3, JUEVES=4;
    int VIERNES=5, SABADO=6, DOMINGO=7;
    String[] NOMBRES_DIAS = {"", "lunes", "martes", "miércoles",
                             "jueves", "viernes", "sábado", "domingo"};
}
```

Esta definición nos permite utilizar las constantes declaradas en cualquier clase que implemente la interfaz, de manera tan simple como:

```
System.out.println (DiasSemana.NOMBRES_DIAS[LUNES]);
```

6.1. Ejemplo de diseño de interfaz e implementación en una clase

Se presentan un par de interfaces que incorporan datos (de tipo primitivo y de referencia en clase) y métodos y una clase que las implementa. En la declaración de la clase se ve que sólo implementa la interfaz `B`, pero como esta interfaz deriva de la interfaz `A` resulta que la clase está implementando las dos interfaces.

```
package UD05;

import java.util.Date;

interface A {
    Date ULTIMA_CREACION = new Date(0, 0, 1);
    void metodoA();
}

interface B extends A {
    int VALOR_B = 20;
    // 1 -1 -1900
    void metodoB();
}

public class Interfaces implements B {

    private long b;
    private Date fechaCreacion = new Date();

    public Anexo6Interfaces(int factor) {
        b = VALOR_B * factor;
        ULTIMA_CREACION.setTime(fechaCreacion.getTime());
    }

    public void metodoA() {
```

```
27          System.out.println("En metodoA, ULTIMA_CREACION = " + ULTIMA_CREACION);
28      }
29
30      public void metodoB() {
31          System.out.println("En metodoB, b = " + b);
32      }
33
34      public static void main(String[] args) {
35          System.out.println("Inicialmente, ULTIMA_CREACION = " + ULTIMA_CREACION);
36          Interfaces obj = new Interfaces(5);
37          obj.metodoA();
38          obj.metodoB();
39          A pa = obj;
40          B pb = obj;
41      }
42  }
```

Si lo ejecutamos obtendremos:

```
1  Inicialmente, ULTIMA_CREACION = Mon Jan 01 00:00:00 CET 1900
2  En metodoA, ULTIMA_CREACION = Thu Aug 26 16:09:47 CEST 2021
3  En metodoB, b = 100
```

El ejemplo sirve para ilustrar algunos puntos:

- Comprobamos que los datos miembro de las interfaces son `static`, ya que en el método `main()` hacemos referencia al dato miembro `ULTIMA_CREACION` sin indicar ningún objeto de la clase.
- Si hubiéramos intentado modificar los datos `VALOR_B` o `ULTIMA_CREACION` no habríamos podido porque es final, pero en cambio sí podemos modificar el contenido del objeto `Date` apuntado por `ULTIMA_CREACION`, que corresponde al momento temporal de la última creación de un objeto ya cada nueva creación se actualiza su contenido.
- En las dos últimas instrucciones del método `main()` vemos que podemos declarar variables `pa` y `pb` de las interfaces y utilizarlas para hacer referencia a objetos de la clase `EjemploInterfaz()`.

7. Ejemplo Anexo UD05

7.1. Anexo1Wrappers

```java
package UD05;

public class Anexo1Wrappers {

    public static void main(String[] args) {

        // WRAPPERS
        Integer i1 = new Integer(42);
        Integer i2 = new Integer("42");
        Float f1 = new Float(3.14f);
        Float f2 = new Float("3.14f");

        Integer y = new Integer(567);      //Crea el objeto
        y++;                    //Lo desenvuelve, incrementa y lo vuelve a envolver
        System.out.println("Valor: " + y); //Imprime el valor del Objeto y

        // VALUEOF
        // Convierte el 101011 (base 2) a 43 y le asigna el valor al objeto Integer i1
        Integer i3 = Integer.valueOf("101011", 2);
        System.out.println(i3);

        // Asigna 3.14 al objeto Float f2
        Float f3 = Float.valueOf("3.14f");
        System.out.println(f3);

        // XXXVALUE
        Integer i4 = 120; // Crea un nuevo objeto wrapper
        byte b = i4.byteValue(); // Convierte el valor de i2 a un primitivo byte
        short s1 = i4.shortValue(); // Otro de los métodos de Integer
        double d = i4.doubleValue(); // Otro de los métodos xxxValue de Integer
        System.out.println(s1); // Muestra 120 como resultado

        Float f4 = 3.14f; // Crea un nuevo objeto wrapper
        short s2 = f4.shortValue(); // Convierte el valor de f2 en un primitivo short
        System.out.println(s2); // El resultado es 3 (truncado, no redondeado)

        // PARSEXXXX
        double d4 = Double.parseDouble("3.14"); // Convierte un String a primitivo
        System.out.println("d4 = " + d4);   // El resultado será d4 = 3.14
        long l2 = Long.parseLong("101010", 2);  // un String binario a primitivo
        System.out.println("l2 = " + l2);   // El resultado es L2 42

        // TOSTRING
        Double d1 = new Double("3.14");
        System.out.println("d1 = " + d1.toString()); // El resultado es d = 3.14
        String d2 = Double.toString(3.14); // d2 = "3.14"
        System.out.println("d2 = " + d2); // El resultado es d = 3.14
        String s3 = "hex = " + Long.toString(254, 16); // s = "hex = fe"
        System.out.println("s3 = " + s3); // El resultado es d = 3.14
```

```
50
51          // TOXXXSTRING
52          String s4 = Integer.toHexString(254); // Convierte 254 a hex
53          System.out.println("254 es " + s4); // Resultado: "254 es fe"
54          String s5 = Long.toOctalString(254); // Convierte 254 a octal
55          System.out.println("254(oct) = " + s5); // Resultado: "254(oct) = 376"
56      }
57  }
```

7.2. Anexo2Date

```
1   package UD05;
2
3   import java.util.Calendar;
4   import java.util.Date;
5   import java.util.GregorianCalendar;
6   import java.time.*;
7   import java.time.format.DateTimeFormatter;
8   import java.time.temporal.ChronoUnit;
9
10  public class Anexo2Date {
11
12      public static void main(String[] args) {
13
14          //Clase Date (java.util.Date)
15          Date fecha = new Date(2021, 8, 19);
16          System.out.println(fecha);           //Mon Sep 19 00:00:00 CEST 3921
17          System.out.println(fecha.getTime()); //61590146400000
18
19          //Clase GregorianCalendar (java.util.Calendar y
    java.util.GregorianCalendar)
20          Calendar calendar = new GregorianCalendar(2021, 8, 19);
21          System.out.println(calendar.getTime()); //Sun Sep 19 00:00:00 CEST 2021
22
23          Date d = new Date();
24          GregorianCalendar c = new GregorianCalendar();
25          System.out.println("Fecha: " + d);  //Fecha: Thu Aug 19 20:06:14 CEST 2021
26          System.out.println("Info: " + c); //Info:
    java.util.GregorianCalendar[time=1629396374723,
27
     //areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.Zone
    Info
28
     //[id="Europe/Madrid",offset=3600000,dstSavings=3600000,useDaylight=true,transiti
    ons=163,
29
     //lastRule=java.util.SimpleTimeZone[id=Europe/Madrid,offset=3600000,dstSavings=36
    00000,
30
     //useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWee
    k=1,
31
     //startTime=3600000,startTimeMode=2,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1
    ,
```

```
 //endTime=3600000,endTimeMode=2]],firstDayOfWeek=2,minimalDaysInFirstWeek=4,ERA=1
,
 //YEAR=2021,MONTH=7,WEEK_OF_YEAR=33,WEEK_OF_MONTH=3,DAY_OF_MONTH=19,DAY_OF_YEAR=2
31,
 //DAY_OF_WEEK=5,DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=8,HOUR_OF_DAY=20,MINUTE=6,SEC
OND=14,
        //MILLISECOND=723,ZONE_OFFSET=3600000,DST_OFFSET=3600000]
        c.setTime(d);
        System.out.print(c.get(Calendar.DAY_OF_MONTH));
        System.out.print("/");
        System.out.print(c.get(Calendar.MONTH) + 1);
        System.out.print("/");
        System.out.println(c.get(Calendar.YEAR) + 1); //19/8/2022

        //LocalDate, LocalTime, LocalDateTime, Duration y Period (java.time.*)
        //LocalDate d = new LocalDate(); //NO compila
        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
        date = LocalDate.now();
        System.out.println(date); //2021-08-19

        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
        time = LocalTime.now();
        System.out.println(time); //20:13:53.118044

        LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35);

        LocalDate date2 = LocalDate.of(1989, 11, 11);
        LocalTime time2 = LocalTime.of(5, 30, 45, 35);
        LocalDateTime dateTime1 = LocalDateTime.of(date, time);
        LocalDateTime dateTime2 = LocalDateTime.now();

        LocalTime localTime1 = LocalTime.of(12, 25);
        LocalTime localTime2 = LocalTime.of(17, 35);
        Duration duration1 = Duration.between(localTime1, localTime2);
        System.out.println(duration1); //PT5H10M
        System.out.println(duration1.toDays()); //0

        LocalDateTime localDateTime1 = LocalDateTime.of(2016, Month.JULY, 18, 14,
13);
        LocalDateTime localDateTime2 = LocalDateTime.of(2016, Month.JULY, 20, 12,
25);
        Duration duration2 = Duration.between(localDateTime1, localDateTime2);
        System.out.println(duration2); //PT46H12M
        System.out.println(duration2.toDays()); //1

        Duration duracion3 = Duration.ofDays(1);
        System.out.println(duracion3); //PT24H
        System.out.println(duracion3.toDays()); //1
```

```
82
83          LocalDate localDate1 = LocalDate.of(2016, 7, 18);
84          LocalDate localDate2 = LocalDate.of(2016, 7, 20);
85          Period periodo1 = Period.between(localDate1, localDate2);
86          System.out.println(periodo1); //P2D
87
88          Period periodo2 = Period.of(1, 2, 3);
89          System.out.println(periodo2); //P1Y2M3D
90
91          Period periodo3 = Period.ofYears(1);
92          System.out.println(periodo3); //P1Y
93
94          //CHRONOUNIT (java.time.temporal.ChronoUnit)
95          LocalDate fechaInicio = LocalDate.of(2016, 7, 18);
96          LocalDate fechaFin = LocalDate.of(2016, 7, 20);
97          // Calculamos el tiempo transcurrido entre las dos fechas
98          // con la clase ChronoUnit y la unidad temporal en la que
99          // queremos que nos lo devuelva, en este caso DAYS.
100         long tiempo = ChronoUnit.DAYS.between(fechaInicio, fechaFin);
101         System.out.println(tiempo); //2
102
103         //Introducir fecha por teclado (java.time.format.DateTimeFormatter)
104         DateTimeFormatter formato = DateTimeFormatter.ofPattern("d/MM/yyyy");
105         String fechaCadena = "16/08/2016";
106         LocalDate mifecha = LocalDate.parse(fechaCadena, formato);
107         System.out.println(formato.format(mifecha)); //16/08/2016
108
109         //Manipulación
110         LocalDate fec = LocalDate.of(2016, 7, 25);
111         LocalDate fec1 = fec.withYear(2017);
112         LocalDate fec2 = fec.withMonth(8);
113         LocalDate fec3 = fec.withDayOfMonth(27);
114         System.out.println(date);  //2016-07-25
115         System.out.println(fec1); //2017-07-25
116         System.out.println(fec2); //2016-08-25
117         System.out.println(fec3); //2016-07-27
118
119         LocalTime tim = LocalTime.of(14, 30, 35);
120         LocalTime tim1 = tim.withHour(20);
121         LocalTime tim2 = tim.withMinute(25);
122         LocalTime tim3 = tim.withSecond(23);
123         LocalTime tim4 = tim.withNano(24);
124         System.out.println(tim);  //14:30:35
125         System.out.println(tim1); //20:30:35
126         System.out.println(tim2); //14:25:35
127         System.out.println(tim3); //14:30:23
128         System.out.println(tim4); //14:30:35.000000024
129
130         LocalDateTime dateTim = LocalDateTime.of(2016, 7, 25, 22, 11, 30);
131         LocalDateTime dateTim1 = dateTim.withYear(2017);
132         LocalDateTime dateTim2 = dateTim.withMonth(8);
133         LocalDateTime dateTim3 = dateTim.withDayOfMonth(27);
134         LocalDateTime dateTim4 = dateTim.withHour(20);
135         LocalDateTime dateTim5 = dateTim.withMinute(25);
136         LocalDateTime dateTim6 = dateTim.withSecond(23);
137         LocalDateTime dateTim7 = dateTim.withNano(24);
138         System.out.println(dateTim);  //2016-07-25T22:11:30
139         System.out.println(dateTim1); //2017-07-25T22:11:30
```

```
        System.out.println(dateTim2); //2016-08-25T22:11:30
        System.out.println(dateTim3); //2016-07-27T22:11:30
        System.out.println(dateTim4); //2016-07-25T20:11:30
        System.out.println(dateTim5); //2016-07-25T22:25:30
        System.out.println(dateTim6); //2016-07-25T22:11:23
        System.out.println(dateTim7); //2016-07-25T22:11:30.000000024

        //OPERACIONES
        LocalDate date3 = LocalDate.of(2016, 7, 18);
        LocalDate date3PlusOneDay = date3.plus(1, ChronoUnit.DAYS);
        LocalDate date3MinusOneDay = date3.minus(1, ChronoUnit.DAYS);
        System.out.println(date3);        // 2016-07-18
        System.out.println(date3PlusOneDay);  // 2016-07-19
        System.out.println(date3MinusOneDay); // 2016-07-17

        LocalDate date4 = LocalDate.of(2016, 7, 18);
        LocalDate date4PlusOneDay = date4.plus(Period.ofDays(1));
        LocalDate date4MinusOneDay = date4.minus(Period.ofDays(1));
        System.out.println(date4);        // 2016-07-18
        System.out.println(date4PlusOneDay);  // 2016-07-19
        System.out.println(date4MinusOneDay); // 2016-07-17

        LocalDate date5 = LocalDate.of(2016, 7, 18);
        LocalDate date5PlusOneDay = date5.plusDays(1);
        LocalDate date5MinusOneDay = date5.minusDays(1);
        System.out.println(date5);        // 2016-07-18
        System.out.println(date5PlusOneDay);  // 2016-07-19
        System.out.println(date5MinusOneDay); // 2016-07-17

        LocalTime time3 = LocalTime.of(15, 30);
        LocalTime time3PlusOneHour = time3.plus(1, ChronoUnit.HOURS);
        LocalTime time3MinusOneHour = time3.minus(1, ChronoUnit.HOURS);
        System.out.println(time3);      // 15:30
        System.out.println(time3PlusOneHour);   // 16:30
        System.out.println(time3MinusOneHour);  // 14:30

        LocalTime time4 = LocalTime.of(15, 30);
        LocalTime time4PlusOneHour = time4.plus(Duration.ofHours(1));
        LocalTime time4MinusOneHour = time4.minus(Duration.ofHours(1));
        System.out.println(time4);      // 15:30
        System.out.println(time4PlusOneHour);   // 16:30
        System.out.println(time4MinusOneHour);  // 14:30

        LocalTime time5 = LocalTime.of(15, 30);
        LocalTime time5PlusOneHour = time5.plusHours(1);
        LocalTime time5MinusOneHour = time5.minusHours(1);
        System.out.println(time5);      // 15:30
        System.out.println(time5PlusOneHour);   // 16:30
        System.out.println(time5MinusOneHour);  // 14:30

        LocalDateTime dateTime3 = LocalDateTime.of(2016, 7, 28, 14, 30);
        LocalDateTime dateTime4 = dateTime3.plus(1, ChronoUnit.DAYS).plus(1,
ChronoUnit.HOURS);
        LocalDateTime dateTime5 = dateTime3.minus(1, ChronoUnit.DAYS).minus(1,
ChronoUnit.HOURS);
        System.out.println(dateTime3); // 2016-07-28T14:30
        System.out.println(dateTime4); // 2016-07-29T15:30
        System.out.println(dateTime5); // 2016-07-27T13:30
```

```
196
197         LocalDateTime dateTime6 = LocalDateTime.of(2016, 7, 28, 14, 30);
198         LocalDateTime dateTime7 =
    dateTime6.plus(Period.ofDays(1)).plus(Duration.ofHours(1));
199         LocalDateTime dateTime8 =
    dateTime6.minus(Period.ofDays(1)).minus(Duration.ofHours(1));
200         System.out.println(dateTime6); // 2016-07-28T14:30
201         System.out.println(dateTime7); // 2016-07-29T15:30
202         System.out.println(dateTime8); // 2016-07-27T13:30
203
204         LocalDateTime dateTime9 = LocalDateTime.of(2016, 7, 28, 14, 30);
205         LocalDateTime dateTime10 = dateTime9.plusDays(1).plusHours(1);
206         LocalDateTime dateTime11 = dateTime9.minusDays(1).minusHours(1);
207         System.out.println(dateTime9);  // 2016-07-28T14:30
208         System.out.println(dateTime10); // 2016-07-29T15:30
209         System.out.println(dateTime11); // 2016-07-27T13:30
210
211         LocalDate dat1 = LocalDate.of(2016, 7, 28);
212         LocalDate dat2 = LocalDate.of(2016, 7, 29);
213         boolean isBefore = dat1.isBefore(dat2); //true
214         boolean isAfter = date2.isAfter(dat1); //true
215         boolean isEqual = dat1.isEqual(dat2); //false
216
217         //Formatos (java.time.format.DateTimeFormatter)
218         LocalDate mifecha2 = LocalDate.of(2016, 7, 25);
219         String fechaTexto = mifecha2.format(DateTimeFormatter.
220                                             ofPattern("eeee',' dd 'de' MMMM 'del'
    yyyy"));
221         System.out.println("La fecha es: " +
222                            fechaTexto); // La fecha es: lunes, 25 de julio del
    2016
223
224         //DAYOFWEEK
225         LocalDate lafecha = LocalDate.of(2016, 7, 25);
226         if (lafecha.getDayOfWeek().equals(DayOfWeek.SATURDAY)) {
227             System.out.println("La fecha es Sábado");
228         } else {
229             System.out.println("La fecha NO es Sábado");
230         }
231         //La fecha NO es Sábado
232     }
233 }
```

7.3. Anexo3Casting

```
1  package UD05;
2
3  public class Anexo3Casting {
4
5      public static void main(String[] args) {
6          // Casting Implicito
7          Persona encargadoCarniceria = new Encargado("Rosa Ramos", 0, null);
8          // Casting Explicito
9          Encargado miEncargado = (Encargado) encargadoCarniceria;
10         miEncargado.setSueldoBase(1200);
11         miEncargado.setSeccion("Carniceria");
```

```
        // Muestra los datos de la Persona que es Encargado
        System.out.println(encargadoCarniceria.toString());

        // Muestra los datos de la Persona que es Encargado
        Empleado empleadoCarniceria = new Empleado("Rosa Ramos", 0);
        System.out.println(empleadoCarniceria instanceof Encargado); //false
        System.out.println(empleadoCarniceria.getClass()); //class Empleado
        System.out.println(empleadoCarniceria.getClass().getSuperclass()); //class
Persona
    }
}
```

7.3.1. `Persona`

```
package UD05;

// Clase Persona que solo dispone de nombre
public class Persona {

    String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    @Override
    public String toString() {
        return "Nombre: " + nombre + "\n";
    }
}
```

7.3.2. `Empleado`

```
package UD05;

// Clase Empleado que hereda de Persona y añade atributo sueldoBase
public class Empleado extends Persona {

    double sueldoBase;

    public Empleado(String nombre, double sueldoBase) {
        super(nombre);
        this.sueldoBase = sueldoBase;
    }

    public double getSueldo() {
        return sueldoBase;
```

```
    }

    public void setSueldoBase(double sueldoBase) {
        this.sueldoBase = sueldoBase;
    }

    @Override
    public String toString() {
        return super.toString() + "Sueldo Base: " + sueldoBase + "\n";
    }
}
```

7.3.3. `Encargado`

```
package UD05;

// Clase Encargado que hereda de Empleado y añade atributo seccion
public class Encargado extends Empleado {

    String seccion;

    public Encargado(String nombre, double sueldoBase, String seccion) {
        super(nombre, sueldoBase);
        this.seccion = seccion;
    }

    public String getSeccion() {
        return seccion;
    }

    public void setSeccion(String seccion) {
        this.seccion = seccion;
    }

    @Override
    public String toString() {
        return super.toString() + "Sección:" + seccion + "\n";
    }
}
```

7.4. `Anexo4ClasesAnidadas`

```
package UD05;

class Pc {

    double precio;

    public String toString() {
        return "El precio del PC es " + this.precio;
    }

    class Monitor {

        String marca;
```

```
14
15          public String toString() {
16              return "El monitor es de la marca " + this.marca;
17          }
18      }
19
20      class Cpu {
21
22          String marca;
23
24          public String toString() {
25              return "La CPU es de la marca " + this.marca;
26          }
27      }
28  }
29
30  public class Anexo5ClasesAnidadas {
31
32      public static void main(String[] args) {
33          Pc miPc = new Pc();
34          Pc.Monitor miMonitor = miPc.new Monitor();
35          Pc.Cpu miCpu = miPc.new Cpu();
36          miPc.precio = 1250.75;
37          miMonitor.marca = "Asus";
38          miCpu.marca = "Acer";
39          System.out.println(miPc); //El precio del PC es 1250.75
40          System.out.println(miMonitor); //El monitor es de la marca Asus
41          System.out.println(miCpu); //La CPU es de la marca Acer
42      }
43  }
```

7.5. `Anexo5Interfaces`

```
1   package UD05;
2
3   import java.util.Date;
4
5   interface A {
6       Date ULTIMA_CREACION = new Date(0, 0, 1);
7       void metodoA();
8   }
9
10  interface B extends A {
11      int VALOR_B = 20;
12      // 1 -1 -1900
13      void metodoB();
14  }
15
16  public class Anexo6Interfaces implements B {
17
18      private long b;
19      private Date fechaCreacion = new Date();
20
21      public Anexo6Interfaces(int factor) {
22          b = VALOR_B * factor;
23          ULTIMA_CREACION.setTime(fechaCreacion.getTime());
```

```
24      }
25
26      public void metodoA() {
27          System.out.println("En metodoA, ULTIMA_CREACION = " + ULTIMA_CREACION);
28      }
29
30      public void metodoB() {
31          System.out.println("En metodoB, b = " + b);
32      }
33
34      public static void main(String[] args) {
35          System.out.println("Inicialmente, ULTIMA_CREACION = " + ULTIMA_CREACION);
36          Anexo6Interfaces obj = new Anexo6Interfaces(5);
37          obj.metodoA();
38          obj.metodoB();
39          A pa = obj;
40          B pb = obj;
41      }
42  }
```

8. Fuentes de información

- Wikipedia
- Programación (Grado Superior) - Juan Carlos Moreno Pérez (Ed. Ra-ma)
- Apuntes IES Henri Matisse (Javi García Jimenez?)
- Apuntes AulaCampus
- Apuntes José Luis Comesaña
- Apuntes IOC Programació bàsica (Joan Arnedo Moreno)
- Apuntes IOC Programació Orientada a Objectes (Joan Arnedo Moreno)