Programación

Bloque 04 - Desarrollo de Clases

Índice

1.- Introducción

Este tema se ha separado en dos partes por necesidades de evaluación. En esta segunda parte vamos a repasar los conceptos básicos de clases y vamos a seguir ampliando estos conceptos. En temas posteriores seguiremos ampliando estos conceptos para tener un mayor conocimiento de la programación.

En esta parte del tema vamos a trabajar con la herencia entre clases, las cuales nos van a servir para reutilizar el código que hemos utilizado en la primera parte del tema.

En esta parte del tema vamos a trabajar con tres librerías de Java muy utilizadas en general y que es necesario de conocer, las clases String, Enum y Fecha.

2.- Repaso de conceptos básicos

Vamos a realizar un pequeño repaso de los conceptos básicos aprendidos en la primera parte del tema:

- **Objeto**: Es una representación en el ordenador de algo que puede existir en el mundo real. Un objeto tiene identidad, el estado y comportamiento.
    - La **identidad** define que todo objeto en el sistema es distinguible de otro objeto en el sistema, aunque puedan parecer idénticos.
    - El **estado** define las propiedades que tiene un objeto, junto con los valores actuales de los mismos.
    - El **comportamiento** define las acciones que puede realizar un objeto a requerimiento de otros objetos del sistema.

- **Clase**: Los objetos de un programa se definen mediante *clases*. Una clase es un grupo de objetos que comparten características comunes. Específicamente tienen el mismo conjunto de atributos (aunque los valores normalmente cambiarán de un objeto a otro) y el mismo comportamiento, esto es, todos ofrecen el mismo conjunto de acciones a los otros objetos del sistema aunque, obviamente, el comportamiento exacto de todos no será exactamente igual ya que dependerá, entre otras cosas, del estado que tenga el objeto, el cual puede cambiar a lo largo del tiempo. Una clase se compone por:
    - **Atributos**. Son las propiedades que tienen los objetos. Cada atributo tendrá un tipo determinado.
    - **Constructores**. Son métodos especiales que se invocan cuando se construyen nuevos objetos de una clase y sirven para inicializarlos.
    - **Métodos**. Son trozos de código que sirven para responder a los mensajes que reciben los objetos de la clase.

2.1.- Visibilidad de los miembros de una clase

Una de las piezas centrales de la programación orientada a objetos es el concepto de

interfaz de una clase. Este concepto no debe de confundirse con las interfaces, que daremos en temas posteriores.

El interfaz de una clase es lo que objetos de esta clase ofrecen al resto de objetos *de otras clases* que forman parte del sistema.

A fin de hacer un buen uso de la programación orientada a objetos hay que procurar que el interfaz de una clase cambie lo menos posible (idealmente nunca), al mismo tiempo que se pueda modificar la implementación cuando y como se quiera.

Java proporciona un mecanismo, llamado ***control de visibilidad*** que permite especificar, para cada miembro de una clase, si forma parte o no del interfaz, o lo que es lo mismo, si el miembro puede ser utilizado desde otras clases o no.

El mecanismo de control de visibilidad es el siguiente:

- **public**. Indica que el miembro es público, esto es, perteneciente al interfaz. Todas las otras clases del sistema pueden acceder libremente al miembro en cuestión.
- **private**. Indica que el miembro es privado, esto es, que sólo es visible para otros miembros de la misma clase y que ninguna otra clase del sistema puede acceder a él, incluyendo clases que hereden de la clase en la que se define el miembro.
- **protected**. Indica que el miembro es protegido, esto es, que sólo es visible para otros miembros de la misma clase y para las clases que hereden de aquella en que se define. El resto de clases del sistema no pueden acceder al miembro.
- En el caso que no se indique nada, se refiere a visibilidad a nivel de paquete, en la que el miembro es accesible a la clase en que se define y por todas las clases contenidas en el mismo paquete que aquella. Este tipo de definición no es muy recomendable.

2.2.- Métodos

Los métodos son bloques de código que se definen para responder a un mensaje que pueda recibir el objeto (o clase, en caso de métodos estáticos).

Un método se define utilizando la siguiente sintaxis:

```
visibilidad [static] tipo nombre(lista_parametros) {
  ..... bloque de instrucciones .....
}
```

donde:

- **visibilidad** es la visibilidad del método. Se usan los mismos modificadores y se aplican las mismas reglas que para la visibilidad de los atributos. La visibilidad es opcional y si no se indica significa visibilidad a nivel de paquete.
- **static** se debe indicar cuando el método no es a nivel de instancia sino método de clase (estático)
- **tipo** es el tipo del valor de retorno que devuelve el método. Si el método no va

a devolver ningún valor de retorno hay que utilizar el tipo especial `void` que indica que no se devuelve nada. **El tipo no es opcional**. Si no se va a devolver ningún valor hay que indicarlo mediante el tipo void.

- **nombre** es el nombre del método.
- **lista_parametros** es la lista de parámetros del método. Indica el número de parámetros que se reciben por el método y el tipo y nombre de cada uno. Los distintos elementos de la lista de parámetros se separan por comas (`,`). Si la lista está vacía (no hay parámetros) hay que seguir indicando los paréntesis pero sin incluir nada dentro.

2.3 El modificador static

En Java, la palabra clave static se utiliza para indicar que un miembro (variable, método, bloque o clase interna) pertenece a la clase en sí misma, en lugar de pertenecer a una instancia (objeto) específica.

Ya hemos visto ejemplos con atributos de clase, los cuales no pertenecían a ninguna instancia, perteneciendo a la clase.

Con static, podemos trabajar los siguientes casos:

- Atributos estáticos (Variables de Clase). En este caso solo existe una copia de esa variable para todos los objetos de esa clase.
- Métodos Estáticos. Pueden ejecutarse sin necesidad de crear un objeto de la clase, aunque no pueden acceder a atributos o métodos "no estáticos" (de instancia) directamente, ni usar la palabra this. Se pueden utilizar para:
    - Contar las instancias de la clase o de objetos específicos.
    - Definir constantes.
    - Realizar métodos que no dependan de los valores de los objetos de la clase. Por ejemplo la raíz cuadrada depende solo de los valores de entrada, no del valor de los objetos de la clase.
    - Crear métodos que no haya que crear un objeto de la clase para obtener el resultado.
    - Métodos Factory. Se utilizan para crear o clonar objetos de manera controlada sin necesidad de llamar directamente al constructor con new.
- Bloques Estáticos. Es un bloque de código que se ejecuta una sola vez, justo cuando la clase es cargada por la Máquina Virtual de Java (JVM).
- Clases estáticas. Java permite declarar clases dentro de otras. Si la clase interna es static, puede ser instanciada sin necesidad de una instancia de la clase externa.

¿Por qué public static void main?

El método principal es estático para que la JVM pueda ejecutarlo al iniciar el programa sin tener que crear primero un objeto de la clase que contiene el main.

2.4.- Acceso a los atributos

Todos los métodos de una clase pueden acceder a **todos** los atributos de la misma, sea cual sea su visibilidad o su pertenencia (atributos de clase o atributos de instancia).

La diferencia estriba en la forma en que se acceden a dichos atributos

2.4.1.- Acceso a los atributos desde un método de instancia

Cuando se envía un mensaje a un método de instancia, el objeto sobre el que se ha enviado el mensaje se pasa automáticamente al método, sin necesidad de incorporar el mismo a la lista de parámetros.

Los atributos del objeto que recibe el mensaje se pueden acceder directamente usando los nombres de los atributos. Esto es aplicable tanto a los atributos de instancia como a los de clase o estáticos. La diferencia estriba en que los atributos de clase son compartidos para todas las instancias mientras que los atributos de instancia son distintos para cada instancia. Cada una tiene su propia copia de los atributos.

Supongamos el siguiente ejemplo:

```
public class Persona {
    private String nombre;

    public void imprimeNombre() {
        System.out.println(nombre);
  }
}
```

En este caso puedes ver que dentro de `imprimeNombre` se está utilizando la "variable" `nombre`, que en realidad está accediendo al atributo `nombre` de la instancia de la clase `Persona` que está recibiendo el mensaje.

Sin embargo, hay que tener cuidado ya que en los métodos se produce el fenómeno llamado sobreposición de variables (variable shadowing). Este fenómeno es una consecuencia del diseño de Java que quiere dejar total libertad al programador para usar los nombres de variable locales que desee dentro de un método, incluyendo los nombre de los parámetros que acepta el método. Como la libertad es total puede ocurrir que el nombre de una variable local o de un parámetro coincida con el nombre de un atributo. Cuando esto pasa, en lugar de ocurrir un error, lo que ocurre es que la variable local "tapa" o se "sobrepone" al atributo y si se usa el nombre a secas se está refiriendo a la variable local y no al atributo.

Veamos un ejemplo de esto:

```
public class Persona {
    private String nombre;

    public Persona() {
        nombre = "Anonimo";
    }

    public void cambiaNombre(String nombre) {
        System.out.println(nombre);
  }
}
```

Si creamos un objeto `Persona`, inicialmente su nombre será "Anonimo". Si mandamos el mensaje `cambiaNombre("Agapito")`, lo que podría esperarse es que en el método se imprimiera "Anonimo". Sin embargo lo que se imprime es "Agapito". Esto es así porque el parámetro `nombre` se sobrepone o "tapa" al atributo `nombre` por lo que al acceder usando nombre estamos accediendo al parámetro, no al atributo.

Para remediar esto podemos utilizar una variable llamada `this` que se crea automáticamente sin necesidad de intervención del programador y que podemos utilizar dentro del método. Esta variable `this` es una referencia al objeto que ha recibido el mensaje y la podemos utilizar para acceder a los atributos del objeto usando la sintaxis `this.atributo`.

Ahora en nuestro código podriamos hacer:

```
public class Persona {
    private String nombre;

    public Persona() {
        nombre = "Anonimo";
    }

    public void cambiaNombre(String nombre) {
        System.out.println(nombre);
        System.out.println(this.nombre);
    }
}
```

Que mostrará por pantalla "Agapito" en la primera línea pero "Anonimo" en la segunda. La primera accede al parámetro nombre y la segunda accede al atributo nombre de la instancia que recibe el mensaje.

Si lo que se "tapa" es un atributo estático podemos utilizar la sintaxis habitual `NombreClase.atributo` (preferida) o `this.atributo`.

2.4.2.- Acceso a los atributos desde un método de clase

Los métodos de clase están más limitados respecto al acceso a los miembros de la clase ya que sólo pueden acceder a los atributos estáticos. No tienen acceso a los atributos de instancia ya que no se llaman sobre un objeto sino sobre la clase.

Si se usa un objeto de la propia clase, sin embargo, si tienen acceso a todos sus atributos. Por ejemplo, si dentro de un método estático se crea un objeto de la misma clase, usando `new`, por ejemplo, el método **si** podrá acceder a todos los atributos de este nuevo objeto.

Ejemplo:

```
public class Persona {
    private static nombreClase;
    private String nombre;

    public Persona() {
        nombreClase = "Clase Persona";
        nombre = "Anonimo";
    }

  public static void pruebaEstatica(String nombreClase) {
    System.out.println(nombreClase);
    System.out.println(Persona.nombreClase);

    // Esta línea no funciona:
    System.out.println(nombre);

    // Creamos una instancia y accedemos a su atributo
    nombre Persona persona = new Persona();
    persona.nombre = "Agapito";
    System.out.println(persona.nombre);
  }
}
```

En este ejemplo tenemos que la clase `Persona` tiene un atributo estático, `nombreClase`, y otro de instancia, `nombre`.
En el método estático `pruebaEstática`, que recibe un parámetro, `nombreClase`, intentamos imprimir una serie de cosas.
La primera línea imprimirá lo que sea el valor que se haya pasado por parámetros al método, no el atributo estático con el mismo nombre, ya que el parámetro "tapa" al atributo.
La segunda si imprime el atributo pero tenemos que usar el nombre de la clase para "destaparlo". La tercera no compilará, aunque la hemos dejado para propósitos demostrativos. La línea no funciona porque un método estático no puede acceder a variables de instancia, puesto que se invocan sin mediación de una instancia sino directamente de la clase.
En las dos siguientes lineas se crea un nuevo objeto `Persona` y se le cambia el nombre a "Agapito". Nótese que a pesar de ser un atributo de visibilidad privada, el atributo puede ser accedido perfectamente desde el método estático ya que éste forma parte de la clase. Nótese también que es necesario usar la referencia para acceder al atributo y no se puede usar el nombre sólo.

2.5.- Envío de mensajes dentro de la misma clase

Dentro de una misma clase se pueden realizar envíos de mensajes (de hecho esta es la única razón de ser de los métodos privados). Hay que hacer notar que la situación es diferente en el caso de métodos estáticos o no estáticos.

En el caso de métodos no estáticos, si se quiere enviar otro mensaje al mismo objeto cuyo mensaje se está procesando por el primer método, se puede obviar la parte objeto del envío de mensajes. El objeto que recibirá el mensaje será el mismo que estaba procesando el primer mensje. Cuando se termine el proceso de este segundo se resumirá el proceso del primero.

Ejemplo:

```
public class Persona {

  public String nombre;

  public Persona() {
     nombre = "Anonimo";
  }

  public void metodo1(){
     System.out.println(nombre);
     metodo2();
  }

  private void metodo2() {
     System.out.println(nombre);
  }
}
```

Si hacemos desde otra clase:

```
Persona persona = new Persona();
persona.nombre = "Agapito";
persona.metodo1();
```

veremos que en ambos caso se imprime "Agapito".

Lo que está ocurriendo es que desde el código externo se está enviando el mensaje `metodo1` al objeto cuya referencia está almacenada en `persona`. Esto provoca que se ejecute el método `metodo1` de la clase `Persona`, pasando como objeto el mismo que estaba almacenado en la variable `persona`. Dentro de este método se imprime el valor del atributo `nombre`, que como se cambió anteriormente vale "Agapito". A continuación se envía el mensaje `metodo2`. Dado que no se especifica objeto, el mensaje se envía al mismo objeto en el que estabamos procesando el anterior mensaje (el mismo que sigue almacenado en `persona`). En este caso se llama al método `metodo2`. Este imprime el atributo `nombre` que sigue siendo el mismo y termina. A volver de procesar el mensaje `metodo2`, el método `metodo1` termina también porque se acabó el bloque y se vuelve al código de llamada original.

En el caso de llamada a un método estático en lugar de un método no estático, la diferencia es que el objeto que estaba recibiendo el mensaje no se pasa al método estático (lo cual tiene sentido). Por lo demás es exactamente igual.

En el caso de los métodos estáticos es distinto. Un método estático no puede enviar mensajes al objeto que está procesando el mensaje ***porque no existe tal objeto***. Por lo tanto un método estático no puede enviar un mensaje a un objeto sin especificar el mismo. Si puede, en cambio, usar una referencia para enviar un mensaje o enviar un mensaje sin objeto, al estilo que se hacía en los métodos de instancia. La diferencia es que, en este caso, el método deberá ser también estático o la compilación fallará.

Por ejemplo, el siguiente código:

```
public class Persona {

  public String nombre;

  public Persona() {
     nombre = "Anonimo";
  }

  public static void metodo1() {
     metodo2();
  }

  private void metodo2() {
     System.out.println(nombre);
  }
}
```

fallará ya que al entrar en el `metodo1` se intenta enviar un mensaje `metodo2`. Como este mensaje es para instancias y no se proporciona ninguna, el código no compila.

El siguiente, sin embargo, sin compilará:

```
public class Persona {

  public String nombre;

  public Persona() {
      nombre = "Anonimo";
  }

  public static void metodo1() {
      metodo2();
  }

  private static void metodo2() {
      System.out.println(nombre);
  }
}
```

Ya que `metodo2` también es estático y no requiere una instancia. También funcionaría:

```
public class Persona {

  public String nombre;

  public Persona() {
      nombre = "Anonimo";
  }

  public static void metodo1() {
      Persona p = new Persona();
      p.metodo2();
  }

  private metodo2() {
      System.out.println(nombre);
  }
}
```

En este caso `metodo2` es de instancia pero como se llama usando una instancia (que se crea dentro de `metodo1`), no hay ningún problema.

2.6.- Valores de retorno

Los métodos pueden devolver uno (y sólo uno) valor de retorno al terminar la ejecución del cuerpo. Para ello se debe utilizar la instrucción `return`, de la forma:

```
return valor;
```

donde valor es el valor de retorno a devolver. El valor de retorno debe ser del mismo tipo que el valor declarado como resultado en la cabecera del método o debe poder ser convertible de forma implícita a un valor de dicho tipo.

Un efecto secundario pero importante de la instrucción `return` es que se finaliza ***inmediatamente*** la ejecución del cuerpo del método y se devuelve el resultado. No es necesario que se llegue al final de las instrucciones del cuerpo del método para terminarlo.

Otra característica de la instrucción `return` es que se pueden utilizar tantas como el programador quiera dentro del cuerpo de un método.

Si el método no devuelve ningún valor de retorno, se puede seguir utilizando `return`, sin proporcionar un valor, para terminar de forma inmediata la ejecución del método en cuestión.

El valor de retorno reemplazará en la expresión donde se empleó el envío del mensaje a éste y se empleará para terminar de calcular la expresión, de la misma forma que una variable dentro de una expresión se reemplaza por el contenido de dicha variable. La diferencia aqui es que el paso del mensaje implica ejecutar algún código antes de devolver el valor, a diferencia de una variable en la que la recuperación del valor contenido en la misma es inmediato.

2.7.- Ocultación completa de atributos - Métodos accesores/mutadores

Es una buena práctica en programación orientada a objetos que el interfaz de una clase ofrezca sólo métodos y no ofrezca atributos.

Está claro que una clase que no sea trivial o sea sólo comportamiento requerirá de atributos pero lo que se solicita no es que no tengan atributos sino que estos no sean públicos y por tanto accesibles desde el interfaz. Por lo tanto todos los atributos deberían declararse como privados por defecto y promocionarlos a protegidos o accesibles sólo desde el paquete sólo cuando la ocasión lo requiera.

Sin embargo en muchas ocasiones algunos de los datos contenidos en los atributos son interesantes para los otros objetos del sistema. Supongamos por ejemplo, una clase

`Persona` con un atributo `nombre`, que contiene el nombre de la persona. Este dato seguro que será de utilidad para otros objetos del sistema pero si se declara como privado no será accesible para ellos.

¿Cómo hacemos para solucionar este dilema?

La solución es sencilla: Cuando tengamos datos que queremos hacer accesibles desde otros objetos de otras clases pero queremos mantener el control sobre lo que se hace con ellos, lo que debemos hacer es ofrecer métodos accesores/mutadores.

Los métodos accesores/mutadores (comunmente conocidos por getters/setters, por su forma de nombrarlos, como veremos a continuación) son métodos cuyo propósito es ofrecer acceso o modificar (de ahí el nombre) de forma controlada la información interna contenida en un objeto. Son una solución mucho mejor que el acceso directo a los atributos por varias razones:

- Podemos controlar el modo en que se accede a un dato (sólo lectura, sólo escritura o lectura/escritura).
- Si sustituimos la forma interna en que se representan los datos dentro del objeto puede ocurrir que algunos atributos aparezcan u otros se destruyan. Mientras los métodos accesores sigan proporcionando una forma de acceder a estos datos, para los objetos externos no habrá cambiado nada.

Por convención, los métodos (accesores) que permiten leer un dato se hacen de la forma:

```
public tipo getDato()
```

donde:

- El método debe ser público (si se quiere que se pueda acceder desde fuera debe ser asi)
- `tipo`. Tipo del dato que devuelve, primitivo u objeto.
- `Dato`. Nombre del dato en cuestión.

Por ejemplo, el método para acceder al atributo `nombre` de `Persona` que comentábamos antes se escribiría:

```
public String getNombre()
```

Hay un caso especial en que el nombre cambia. Por tradición, cuando el tipo del dato que se devuelve es boolean, se cambia get por is, de forma que si persona tuviera un atributo casado, que indicara si la persona está casada (true) o no (false), el método para acceder a este dato se escribiría:

```
public boolean isCasado()
```

Los métodos que sirven para modificar (mutadores) un dato siguen la convención siguiente:

```
public void setDato(tipo dato)
```

Donde:

- El método debe ser `public` (porque debe ser accesibles desde otras clases) y `void`
  (porque no devuelve ningún valor sino que sirve para modificarlo).
- `Dato` es el nombre del dato a modificar.
- `tipo` es el tipo del dato (primitivo u objeto)
- `dato` es el nuevo valor del dato.

Por ejemplo, para modificar los dos datos anteriores, los métodos se escribirían de la forma:

```
public void setNombre(String
nombre) public void
```

2.8.- Constructores

Los constructores son un tipo especial de método que se incluyen en una clase y que tienen el propósito específico de inicializar una instancia de un objeto.

Los constructores, por tanto, vienen a solucionar el problema de ajustar el estado de un nuevo objeto a valores consistentes con la definición de dicho objeto.

2.8.1.- Definición de constructores

Los constructores de una clase se definen igual que los métodos de la misma pero con algunas pequeñas diferencias:

El constructor de una clase debe tener como nombre el mismo nombre que la clase.

Un constructor no tiene tipo (ni siquiera void), ya que no puede devolver valores. Tampoco puede ser estático, ya que se emplea para crear una instancia. Se puede emplear `return` dentro de ellos pero sin usar valor.

En cambio, un constructor **si** puede tener modificador de visibilidad. De esta forma podemos tener constructores que son privados y, por lo tanto, no se pueden usar desde otros objetos de otras clases.

Los constructores pueden estar sobrecargados, igual que otros métodos, por lo que podemos tener diferentes constructores con diferentes listas de parámetros.

Si no se define ningún constructor, Java crea uno automáticamente (aunque no podamos ver el código), llamado el constructor por defecto, que no tiene parámetros y que asigna a los atributos los valores por defecto para su tipo:

- Tipos numéricos: Valor 0
- `char`: Valor nulo '\u0000'
- `boolean`: Valor `false`.
- Objetos (y arrays): `null`

Dentro del cuerpo del constructor se define el objeto `this`, que referencia al objeto que se está construyendo.

2.8.2.- Invocación de constructores

Los constructores se llaman automáticamente al usar el operador `new`. A partir del nombre de la clase y del número y valores de los parámetros se elige el método constructor a emplear, de la misma forma que con los métodos sobrecargados "normales".

El constructor se llama después de crear el objeto en memoria pero antes de devolver dicho objeto desde `new` de forma que tiene tiempo de inicializar de forma apropiada el objeto antes de que éste sea usado.

Una cuestión a tener muy en cuenta es la forma en que Java trata a los constructores, que puede producir algunos comportamientos inesperados, aunque no extraños cuando se miran de cerca.

En primer lugar, como ha hemos dicho, la definición de constructores es voluntaria. Un programador puede elegir no definir ningún constructor. En ese caso, como se ha dicho, Java proporciona automáticamente un constructor por defecto "fantasma", que no hace más que inicializar los atributos de la forma que se ha indicado.

Lo interesante es que si el programador **si** decide definir uno o más constructores, es obligatorio usar uno de estos para crear un objeto de la clase en cuestión **y no se crea el constructor por defecto "fantasma"**, aunque el programador tiene la potestad de que uno de los constructores que defina sea el por defecto (sin parámetros).

Este comportamiento, como se indicó antes, provoca algunos comportamientos "curiosos":

- Si se definen constructores pero no el por defecto, no se podrán crear objetos utilizando éste, ya que no existirá. Hay que utilizar alguno de los otros constructores.
- Si se define uno o más constructores pero ninguno es público, **no se podrán crear objetos desde fuera usando new**. Esto es ciertamente curioso pero sigue manteniendo la lógica: Si se definen constructores hay que usar uno de ellos. Si ninguno de ellos es visible, entonces no se podrá usar ninguno, imposibilitando en la práctica la creación de objetos de esa clase mediante new, aunque se pueden buscar vías alternativas, la cual es la razón inicial para bloquear los constructores en primer lugar.

2.8.3.- Invocación de un constructor a otro

A veces es interesante el poder hacer una llamada desde un constructor a otro, de forma que reutilicemos parte del código haciéndolo más eficiente y lo que es más importante, no repitiendo código que después posiblemente haya que cambiar más tarde.

Supongamos el caso de la siguiente clase:

```
public class Persona {
   private String nombre;
   private int edad;
   private int numHijos;
   private double peso;

   // Constructor por defecto
   public Persona() {
      nombre = ""; edad = 0;
      numHijos = 0;
      peso = 0;
   }

   // Constructor con nombre
   public Persona(String nombre) {
      this.nombre = nombre;
      edad = 0;
      numHijos = 0;
      peso = 0;
   }
}
```

Como se puede ver, gran parte del código de los dos constructores es el mismo, concretamente la parte que inicializa a `edad`, `numHijos` y `peso`. ¿Sería posible "reciclar" ese código de forma que se use por los dos constructores, sin necesidad de repetirlo? La respuesta, obviamente es sí.

Si observamos el ejemplo cuidadosamente veremos que si desde el constructor con el parámetro nombre llamamos al por defecto, que inicialice casi todos los atributos como queremos y después cambiamos el nombre y le damos el valor proporcionado por el parámetro efectivamente hacemos lo mismo sin necesidad de duplicar el código. El código quedaría:

```
public class Persona {
   private String nombre;
   private int edad;
   private int numHijos;
   private double peso;

   // Constructor por defecto
   public Persona() {
      nombre = ""; edad = 0;
      numHijos = 0;
      peso = 0;
   }

   // Constructor con nombre
   public Persona(String nombre) {
      // Llamamos al constructor por defecto
      Persona();
      This.nombre = nombre;
   }
}
```

Desafortunadamente esto no funciona porque `Persona` es una clase, no un método y Java protesta y no compila. ¿Significa esto que no podemos hacer esta llamada? No. Lo que ocurre es que debemos hacerla de forma diferente usando `this`, en lugar del nombre de la clase. El siguiente código hace lo que buscábamos y funciona correctamente:

```
public class Persona {
   private String nombre;
   private int edad;
   private int numHijos;
   private double peso;

   // Constructor por defecto
   public Persona() {
      nombre = ""; edad = 0;
      numHijos = 0;
      peso = 0;
   }

  // Constructor con nombre
  public Persona(String nombre) {
    // Llamamos al constructor por defecto
    this();
    this.nombre = nombre;
  }
}
```

Como puedes ver al usar `this` como un nombre de método se invoca al constructor adecuado según la lista de parámetros.

Un problema adicional es que la llamada de un constructor a otro ***debe ser la primer instrucción dentro del constructor, no se puede hacer más adelante dentro del cuerpo***. Esto puede provocar problemas en caso de que haya que hacer comprobaciones o cualquier tipo de cálculo en uno antes de llamar al otro. En este caso, el código común se puede sacar a un método privado.

2.8.4.- Inicialización estática de atributos

Los atributos también se pueden utilizar de la manera a la que estamos acostumbrados a inicializar variables locales, esto es, colocando el signo = y un valor después de la declaración.

Como ya se ha dicho, esto funciona también en Java y podemos utilizarlo cuando queramos. Los constructores dan una forma más avanzada de inicialización porque permite incluir lógica (condiciones, ciclos, etc.) en la misma.

De hecho podemos combinar las dos e inicializar atributos usando la inicialización "normal" asi como constructores. En este caso hay que tener en cuenta que cuando se crea un objeto primero se reserva la memoria, a continuación se realiza la inicialización "normal" y a continuación se invoca el constructor adecuado.

2.8.5.- Atributos constantes

Es posible hacer atributos que sean constantes. Esto significa que a estos atributos sólo se les puede dar valor una sola vez, mediante inicialización estática, y que su valor no se puede modificar a partir de ese momento.

Para hacer esto hay que añadir, antes del tipo, el modificador `final`. Por ejemplo, si creamos una clase llamada `Matematicas` y dentro de ella declaramos:

```
private final double PI = 3.1415926;
```

Debido a que en la mayoría de las ocasiones estos atributos constantes tienen el mismo valor para todas las instancias, es un desperdicio que todas las instancias de la clase lleven una copia del mismo valor, cuando este no va a variar nunca. Es por esta razón que en la mayoría de las ocasiones, los atributos constantes se declaran de clase, o sea estáticos. En nuestro ejemplo el atributo constante quedaría:

```
private static final double PI = 3.1415926;
```

Otra consideración es que en ocasiones una clase debe ***exportar constantes*** de forma que éstas sean accesibles desde fuera de la clase. El caso anterior es un indicativo de ello. Ya que PI es una constante universal es probable que más de una clase tenga necesidad de usarla. Para ello, en lugar de que varias tengan su propia definición, con más o menos precisión (decimales), es más eficiente que se defina en una sola clase y que el resto usen este valor. Para ello simplemente hay que declarar la constante como pública (public). En nuestro caso quedaría finalmente como :

```
public final double PI = 3.1415926;
```

Desde otras clases se podría usar como `Matematicas.PI`

2.9.- Documentación de clases

Completando JavaDoc podemos crear una documentación de cada clase incluyendo la funcionalidad y todos los datos interesantes de ella. Por ejemplo:

```
/**
 * Clase que representa a una Persona.
 * Una persona es alguien que participa en el sistema, ya sea como
 * empleado o como cliente
 * @author Francisco Fernández
 * @version 1.0
 */
public class Persona {

}
```

Para los métodos (incluyendo constructores, getter y setter y resto de métodos) debemos de hacer algo similar a las funciones:

```
/**
 * Calcula la media de tres valores
 * @param valor1 Primer valor
 * @param valor2 Segundo valor
 * @param valor3 Tercer valor
 * @return Media de los tres valores
 * @author Francisco Fernández
 */
public double calculaMedia3(double valor1, double valor2, double
  valor3) {
    return (valor1 + valor2 + valor3) / 3.0;
}
```

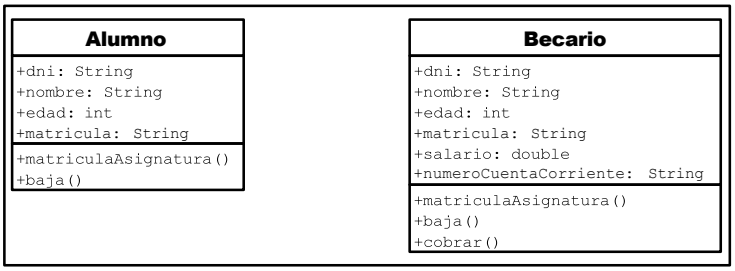
Con ello, ya tendremos nuestro código comentado.

3.- Herencia

En muchas ocasiones durante el desarrollo de una aplicación o librería descubriremos que hay clases que son muy similares, tanto en estado como en comportamiento pero que no son exactamente iguales, sólo parecidas.

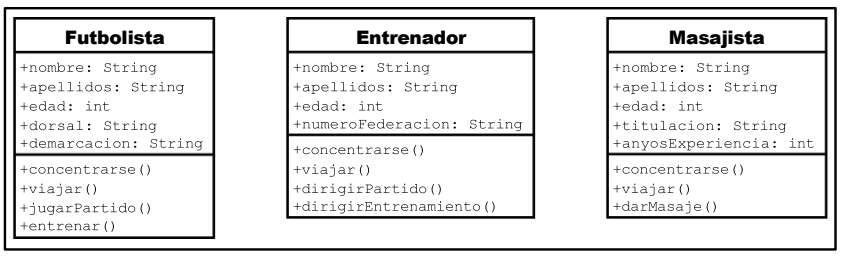
Si examinamos dichas clases en muchas ocasiones, no siempre, percibiremos que la similaridad proviene de que las dos clases realmente están relacionadas en el sentido de que, o bien una de las dos es un tipo especial de la otra, o bien ambas son tipos especiales de una tercera.

Para ver un ejemplo del primer caso, examinemos el siguiente diagrama de clases:



Como se puede ver, ambas clases son muy similares. Si se examinan más a fondo se descubre que un becario tiene todo lo que tiene un alumno y hace todo lo que hace un alumno y además alguna cosa más (salario, numeroCuentaCorriente, etc.). Esto es debido realmente a que un Becario ***es un*** Alumno con alguna cosa más (un becario también es un alumno que además recibe una beca).

Los ejemplos del segundo caso son más complicados de ver pero no mucho. Examinemos ahora otro diagrama de clases:



En este caso no se ve claramente que ninguna clase sea un caso especial de otra pero si se examinan cuidadosamente se puede ver que hay atributos que tienen todas las clases (nombre, apellidos, edad), así como operaciones que también tienen todas (concentrarse, viajar). ¿A qué se debe esto? Esto se debe, principalmente a que cada una de estas clases ***es un*** caso especial de otra clase que no aparece en el diagrama. Esta clase representa una persona que está vinculada con un equipo de futbol, a la cual podríamos llamar, por ejemplo, `MiembroEquipo`. Esta clase tendría los atributos y métodos comunes a las tres clases. De esta forma se podría decir que tanto un `Futbolista`, como un `Entrenador` como un `Masajista` son también `MiembrosEquipo`, lo cual es cierto, aunque esta clase no la viéramos al hacer el análisis inicial del problema.

Al examinar estas relaciones alguno podría pensar que sería bueno el poder aprovechar esta similaridad entre clases para evitar el tener que repetir el mismo código en varias clases y poder poner el código común en algún sitio de forma que todas las que tienen algo en común las usen. El mecanismo para hacer esto existe y se llama ***herencia***, siendo este uno de los pilares fundamentales de la programación orientada a objetos.

Se dice que una clase hereda de otra cuando la primera es un caso especial de la segunda (por ejemplo el caso que teníamos de `Alumno` y `Becario`). En aquél caso, un `Becario` es un caso especial de un `Alumno`. O dicho de otra forma un `Becario` es un `Alumno`, con alguna diferencia o peculiaridad especial. A la clase de la que se hereda se le denomina ***clase padre*** o ***superclase***. A la clase que hereda se le denomina ***clase hija*** o ***subclase***.

Para que la herencia sea real y no apostada o simulada, debe cumplirse la siguiente condición necesaria: **Siempre debe poder utilizarse un objeto de la subclase en cualquier situación en que se pueda utilizar un objeto de la superclase.**

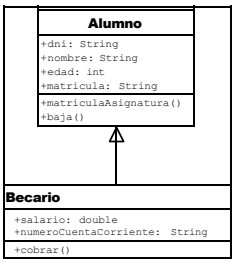
En nuestro ejemplo esta condición implicaría que en cualquier situación en que se podamos utilizar un `Alumno`, también debería poder usarse un `Becario`. Lo complementario no debe porque ser cierto y en la gran mayoría de los casos no lo será.

Si quieres otra versión de la condición, podrías verla como que todo `Becario` es siempre y a la vez un `Alumno` y un `Becario` pero no todo `Alumno` es también un `Becario`. Por lo tanto, en cualquier cosa en que participe un `Alumno` se debería poder usar un `Becario`, puesto que un `Becario` siempre es un `Alumno` pero en una situación en que se necesite un `Becario` no se puede utilizar cualquier `Alumno`, ya que no todos los `Alumno`s son también `Becario`s.

Cuando una subclase hereda de una superclase, adquiere todos los atributos y mensajes de la superclase, pudiendo añadir los propios e incluso modificar, de forma controlada, el comportamiento de los mensajes, aunque esto hay que hacerlo con sumo cuidado para evitar romper la condición anterior.

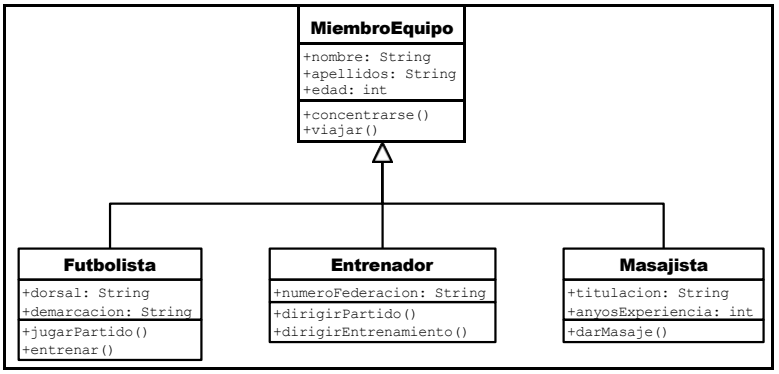
En nuestros ejemplos, los esquemas quedarían como se muestra a continuación. En el

caso del primer esquema:



Aquí se ve como `Becario` hereda de `Alumno`. En `Becario` sólo se representan los atributos y mensajes ***nuevos*** que son propios de esta clase únicamente, aunque también disponga de todos los atributos y mensajes que tiene `Alumno`. De esta forma estos atributos y mensajes comunes se escriben en sólo un sitio (`Alumno`) pero se usan en las dos clases sin necesidad de repetir.

En el caso del segundo esquema:

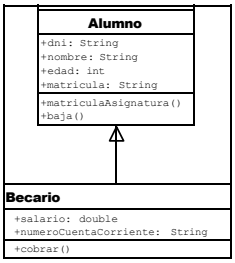


Se puede ver que hemos añadido una nueva clase (`MiembroEquipo`) con los atributos y mensajes comunes a las tres clases previamente existentes (`Futbolista`, `Entrenador` y `Masajista`). En estas clases quedan los atributos y mensajes que son particulares de cada una de estas clases, ya que las comunes las obtienen desde `MiembroEquipo`.

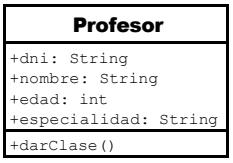
Este concepto se puede extender de forma vertical, es decir, una clase puede heredar de otra clase, que, a su vez, hereda de otra. Este conjunto de relaciones de herencia recuerda a una estructura de árbol, por lo que se llama árbol de herencia. Cuando una subclase hereda de una superclase, hereda no sólo lo que se define explícitamente en la superclase, sino lo que se define en la superclase de ésta y en la superclase de la superclase y así hasta llegar a una clase que no hereda de ninguna otra.

En Java, la clase raíz de todas y la única que no hereda de otras es la clase predefinida `Object`. **Todas** las clases del sistema heredan directa o indirectamente de `Object`, por lo que disponen de los atributos y métodos de la misma, además de los que se vayan declarando en las distintas clases que están en el árbol de herencia desde `Object` hasta llegar a la clase en cuestión.

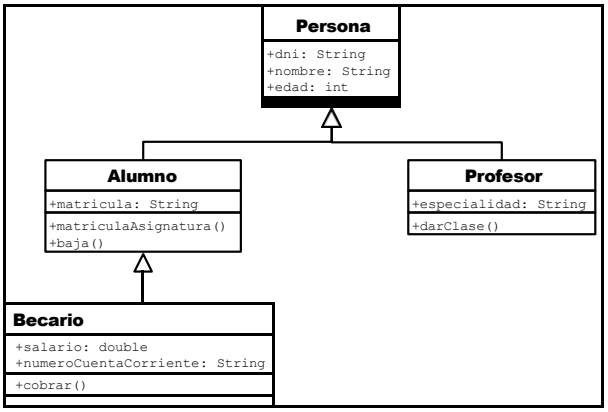
Por ejemplo, retomemos el diagrama con las clases `Alumno` y `Becario`:



Avanzando en el ańalisis de los requisitos, nos damos cuenta de que también debemos incluir una clase para representar a los profesores que tiene la siguiente estructura:



Como puedes ver al compararla con `Alumno`, tienen varias cosas en común (`dni`, `nombre`, `edad`). En este caso podríamos crear una superclase común para `Alumno` y `Profesor` con estos atributos y que ambas los hereden de ella. Si llamamos a esta clase `Persona` (por ejemplo), nuestro diagrama completo quedaría:



En este caso, la clase `Persona` tiene tres atributos (`dni`, `nombre`, `edad`), `Alumno` tiene cuatro: los tres heredados de `Persona` (`dni`, `nombre` y `edad`) más el que define explicitamente (`matricula`). `Profesor` tiene también 4, los tres que hereda de `Persona` (`dni`, `nombre` y `edad`) mas el que define explicitamente (`especialidad`). Ambos definen sus propios métodos de forma explícita, ya que `Persona` no define ninguno. `Becario` hereda `dni`, nombre y `edad` de `Persona` a través de `Alumno`, matricula directamente desde `Alumno` y define dos atributos propios (`salario` y `numeroCuentaCorriente`). Tendrá tres métodos, dos heredados de `Alumno` (`matriculaAsignatura` y `baja`) y uno definido explícitamente (`cobrar`).

En Java una clase sólo puede heredar de una única clase, en constraste con otros lenguajes (como C++, por ejemplo) que permiten lo que se denomina ***herencia múltiple*** que consiste en que una clase puede heredad de más de una.

La herencia múltiple más compleja porque pueden ocurrir "choques" entre los atributos que se heredan por un lado o por otro.

Java sin embargo si permite un cierto tipo de herencia múltiple cuando lo único que se hereda son comportamientos. Ya veremos esto en otro bloque más adelante cundo veamos las instancias.

3.1.- Creación de clases heredadas

La creación de clases heredadas se realiza mediante la instrucción extends, además del nombre de la clase de la que hereda. Si creamos la clase nombre:

```
public class nombre
```

Podemos crear una clase hija de la forma:

```
public class nombre extends padre
```

En nuestro ejemplo tendríamos:

Persona.java

```
public class Persona
  { protected String
  nombre; protected
  String dni;
  protected int
  edad;

  public Persona(String nombre, String dni, int edad) {
    ....
  }
}
```

Alumno.java

```
public class Alumno extends Persona {
   protected String matricula;

   public Alumno(String nombre, String dni, int edad, String  matricula) {
     ....
   }
   public void matriculaAsignatura(String asignatura) {
     .....
   }
   public void baja() {
     ....
   }
}
```

Profesor.java

```
public class Profesor extends Persona {
   protected String especialidad;

   public Profesor(String nombre, String dni, int edad, String especialidad)
   {
     ....
   }
   public void darClase() {
     .....
   }
}
```

Becario.java

```
public class Becario extends Alumno {
   protected double salario;
   protected String numeroCuentaCorriente;

   public Becario(String nombre, String dni, int edad, String
      matricula, double salario, String numeroCuentaCorriente)
   {
     ....
   }
   public void cobrar() {
     .....
   }
}
```

3.1.1.- Constructores en clases heredadas

Los constructores en las subclases pueden utilizar los constructores en las superclases (siempre que la visibilidad de éste último lo permita).

Para invocar desde el constructor de una subclase al de una superclase no se puede emplear el nombre de la clase hay que usar `super`(...).

Por ejemplo, supongamos la implementación de las clases anteriores (Persona y Alumno, por ejemplo). En ellas podríamos usar el constructor de Persona desde la clase Alumno ya que comparten tres atributos y se inician de la misma forma en ambas clases.

Persona.java

```
public class Persona {
   protected String nombre;
   protected String dni;
   protected int edad;

   public Persona(String nombre, String dni, int edad) {
    // Inicializa los atributos
    this.nombre = nombre;
    this.dni = dni;
    this.edad = edad;
  }
}
```

Alumno.java

```
public class Alumno extends Persona {
   protected String matricula;

   public Alumno(String nombre, String dni, int edad, String matricula) {
      // Usa el constructor de Persona para iniciar los primeros atributos
      super(nombre, dni, edad);
      // Y despues inicia el particular (matricula)
      this.matricula = matricula;
  }
  ......
}
```

La clase Persona no necesita ser modificada. Como se puede ver se emplea:

```
// Usa el constructor de Persona para iniciar los primeros atributos
super(nombre, dni, edad);
```

Para invocar al constructor de `Persona` con los atributos propios de `Persona`. A continuación se procede a iniciar los atributos propios de `Alumno` (`matricula`).

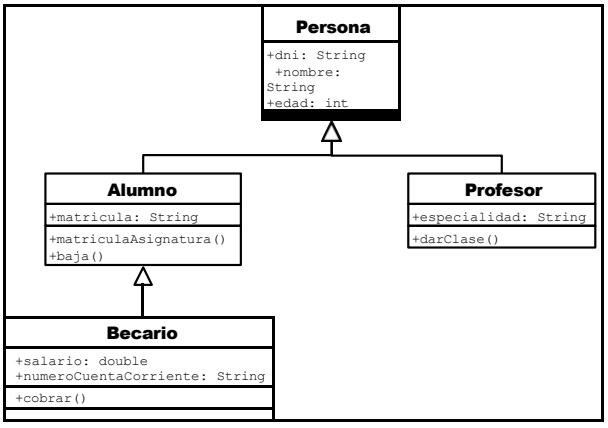
Hay una restricción especial con estas llamadas de constructor de subclase a constructor de superclase, sin embargo. **Es obligatorio que la llamada al constructor de la superclase sea la primera instrucción en la implementación del constructor de la subclase**. Si no se hace así, Java no compila.

3.2.- Uso de clases heredadas

El uso de clases heredadas es bastante simple. La creación se realiza exactamente de la misma manera que vimos en bloques anteriores, usando `new`.

Las diferencias vienen en el uso que se puede hacer de ellas.

Como ejemplos vamos a usar el último esquema que hemos visto de `Persona`, `Alumno`, etc. Vamos a suponer que tenemos estas clases disponibles.



3.2.1.- Sustitución Superclase por Subclase

Esta regla dice que en cualquier lugar donde se pueda emplear un objeto de una superclase, se puede utilizar también un objeto de cualquiera de sus subclases.

Supongamos que tenemos el siguiente código:

```
  Persona persona = new Persona()
```

Nada nuevo aquí que no hayamos visto anteriormente. Lo que sí es nuevo es que también podemos hacer:

```
  Persona persona = new Alumno()
```

o

```
  Persona persona = new Profesor()
```

o incluso

```
  Persona persona = new Becario()
```

Y sería perfectamente válido. Como puedes ver estamos asignando objetos de las subclases a una variable que es de la superclase.

La razón de que esto funcione probablemente la habrás adivinado ya: La regla que dice que cualquier objeto de una superclase debe poder sustituirse por un objeto de una subclase.

Ya que un `Alumno` es una `Persona`, un `Profesor` es una `Persona` y un `Becario` es una persona (indirectamente, pero lo es), es lógico que se pueda asignar un objeto de cualquiera de estos tres tipos a una variable de tipo Persona.

Siguiendo este razonamiento hasta el límite llegamos a la conclusión de que ***cualquier objeto se puede asignar a una variable de tipo `Object`***. Esto es así porque ya hemos dicho que Object es el antecesor último de todas las clases de un sistema. Por lo tanto es superclase, directa o indirectamente, de todas las clases del sistema, así que se puede asignar cualquier objeto a una variable de la clase `Object`.

Esta regla de asignación también aplica en el caso de parámetros a mensajes que no es más que una forma de asignación "diferida" o "remota". En este caso la regla dice que en un parámetro de tipo clase determinado se puede pasar un objeto de esa clase o de cualquiera de sus subclases.

Por ejemplo, supongamos que tenemos una clase `Facultad`, tal que:

```
public class Facultad {
  .......
  public void alistarEnCurso(Alumno alumno, Curso curso) {
    .....
  }
}
```

Desde una clase que creemos nosotros podríamos hacer:

```
// Obtenemos mágicamente una Facultad en la variable facultad
// Y un Curso en la variable nuevoCurso

Alumno alumno = new Alumno();

// Registramos al alumno en el curso
facultad.alistarEnCurso(alumno, nuevoCurso);
```

pero también podríamos hacer:

```
// Obtenemos mágicamente una Facultad en la variable facultad
// Y un Curso en la variable nuevoCurso

Becario becario = new Becario();

// Registramos al alumno en el curso
facultad.alistarEnCurso(becario, nuevoCurso);
```

Ya que un `Becario` también es un `Alumno`.

Es importante tener en cuenta que la instancia realmente es de la subclase. Lo que hacemos con esta asignación es accederla a través de una variable cuyo tipo es de una superclase, pero eso no cambia la naturaleza de la instancia, sólo la forma de accederla.

Si accedemos a un objeto mediante una variable de una clase determinada ***sólo podremos acceder a atributos y mensajes que se puedan utilizar en instancias de dicha clase***. Esto lo veremos a continuación. Supongamos el siguiente código:

```
Alumno alumno1 = new Alumno();
Alumno alumno2 = new Becario();
System.out.println("Nombre del alumno: " + alumno1.nombre);
System.out.println("Nombre del alumno: " + alumno2.nombre);
```

Este código funcionaría bien y mostraría los nombres del alumno y del becario. Este, sin embargo, no funcionaría:

```
Alumno alumno = new Becario();
System.out.println("Nombre del alumno: " + alumno.nombre);
System.out.println("Salario del alumno: " + alumno.salario);
```

La tercera línea no compilaría y Java se quejaría de que "la clase Alumno no tiene un atributo llamado salario). Esto formalmente es cierto ya que la clase no tiene este atributo. El problema es que, usando una variable de una clase determinada sólo podemos acceder a cosas que sabemos seguro que ***todos*** los objetos de dicha clase tienen. No tenemos garantías de que el objeto pueda tener o no atributos de subclases (unas veces sí y otras no). Por lo tanto acceso a atributos y mensajes de subclases están prohibidos, aunque sepamos seguro que una instancia en concreto si los debería tener.

3.2.2.- Conversión forzada (casting)

De la misma forma que esto es automático y legal:

```
Alumno alumno = new Becario();
```

esto no lo es

```
Alumno alumno = new Becario();
Becario becario2 = alumno;
```

En este código creamos un objeto de clase `Becario` y lo asignamos a una referencia de alumno. El objeto es de clase `Becario` pero está referenciado por una variable de clase `Alumno`. Esta línea es OK y funciona bien pero la segunda no compilará.

El problema que nos lanza Java es que no puede convertir un objeto de una superclase a uno de una subclase. Nosotros sabemos que esa conversión puede hacerse ya que el objeto realmente es una instancia de `Becario` pero Java se niega a hacerlo de forma automática ya que no puede asegurar que siempre vaya a ser el caso. Lo mismo ocurre cuando intentamos convertir una referencia a un interfaz a una referencia al objeto de la clase que lo implementa.

Sin embargo si nos proporciona un mecanismo para que podamos hacer esta conversión, dejando en manos del programador la responsabilidad de hacerla y afrontar las consecuencias.

La forma de hacerlo es un viejo conocido nuestro. El operador de conversión de tipo o cast, pero ahora aplicado a clases. El siguiente código ***si*** funcionaría:

```
Alumno alumno = new Becario();
Becario becario2 = (Becario)alumno;
```

En este caso estamos forzando la conversión poniendo el nombre de la clase dentro del operador de conversión.

Esto, sin embargo, fallaría:

```
  Persona persona1 = new Alumno();
  Persona persona2 = new Profesor();
  // Esta funciona
  Alumno alumno1 = (Alumno)persona1;
  // Pero esta no
  Profesor profesor1 = (Profesor)persona1;
```

La última línea falla porque la instancia en persona1 es de la clase `Alumno`, que no es `Profesor` ni una subclase de ésta.

Debes tener cuidado porque esta conversión puede provocar una excepción en caso de que la conversión no se pueda hacer, esto es, que la instancia no sea de la clase que aparece en la conversión o de una de sus subclases. La excepcion que se lanza es de la clase `ClassCastException`.

3.2.3.- El operador instanceof

La conversión forzada discutida en el punto anterior puede dar problemas en caso de que no tengamos claro, en un punto determinado de nuestro programa si esta conversión se puede hacer o no.

Lo ideal sería disponer de algún mecanismo que nos permita el consultar, en tiempo de ejecución, si una referencia puede ser o no convertida de forma segura.

Este mecanismo existe en Java y es el operador `instanceof` (si, es un texto), que funciona de la siguiente forma:

```
instancia instanceof Clase
```

A este operador se le pasa una instancia y una clase y devuelve `true` si el objeto es instancia de la clase o es de una subclase de esta y `false` en caso contrario.

Si ampliamos el último ejemplo:

```
Persona persona1 = new Alumno();
// Se puede comvertir?
System.out.println("Es instancia de Alumno " + persona1 instanceof Alumno);
// Esta funciona
Alumno alumno1 = (Alumno)persona1;
// Se puede comvertir?
System.out.println("Es instancia de Profesor " + persona1 instanceof
Profesor);
// Esta no
Profesor profesor1 = (Profesor)persona1;
```

3.2.4.- Herencia y excepciones

Cuando vimos en bloques anteriores el mecanismo de manejo de excepciones, indicamos que las clausulas `catch` indicaban la clase de la excepción que se capturaba o procesaba en dicho `catch`.

Aunque esto era (y sigue siendo) cierto, la verdad es algo más compleja si incluimos en la mezcla las conversiones entre superclases y subclases que hemos visto en las secciones anteriores.

La regla de los catch la podemos refinar diciendo que un catch captura o procesa una excepción de la clase que se indica como parámetro ***o de cualquiera de sus subclases***. Esto es así porque las excepciones, como todas las clases, pueden participar en el mecanismo de herencia. De hecho es obligatorio que participen, ya que una clase que se quiera usar como excepción debe heredar de `Exception` o de `RuntimeException` (como vimos en la sección correspondiente usando la palabra `extends`)

Por lo tanto las excepciones heredan unas de otras y si ponemos en un catch una clase de excepción cualquiera esa cláusula capturará esa excepción y sus subclases.

Esto tiene algunas implicaciones curiosas en el sentido de que podemos utilizar este hecho para hacer capturas más o menos específicas de excepciones.

Por ejemplo, cuando trabajemos con ficheros veremos que muchas de los mensajes que se procesan por clases que trabajan con los mismos pueden lanzar una excepción de la clase `java.io.IOException`. Pero además, de esta heredan muchas otras excepciones para indicar errores más concretos. Por ejemplo `java.io.FileNotFoundException` es una excepción que indica que un fichero involucrado en una operación que se ha solicitado no se encuentra.

Nosotros, en nuestro código podremos elegir entre capturar `IOException` en caso de que nos de un poco igual el error que ocurra pero podemos utilizar `FileNotFoundException` en caso de que queramos actuar concretamente cuando no se encuentre un fichero.

```
try {
  // Abre el fichero en modo lectura
  // Puede lanzar FileNotFoundException
  FileInputStream is = new FileInputStream("mifichero");
  // Lee del fichero
  // Puede lanzar
  IOException int
  dato = is.read();
} catch (FileNotFoundException e) {
  System.out.println("Fichero no encontrado");
```

Como los bloques `catch` se van comprobando en el mismo orden en que aparecen en el código, el primero en el que pueda encajar el objeto lanzado es el que se procesará la excepción. Esto significa que si se quiere que funcione bien, hay que incluir las clases más generales (más superclases si se quiere ver así) en los últimos `catch` o, por lo menos, antes que los más específicos (más subclases). Si no éstos últimos serán inservibles porque nunca se tomarán.

Por ejemplo, el caso siguiente es correcto:

```
} catch (IOException e) {
    System.out.println("Error leyendo del fichero");
}
```

Este código mostrará el error "Fichero no encontrado" cuando no se pueda abrir el fichero y "Error leyendo del fichero" si no se puede leer, pero el siguiente:

```
try {
  // Abre el fichero en modo lectura
  // Puede lanzar FileNotFoundException
  FileInputStream is = new FileInputStream("mifichero");
  // Lee del fichero
  // Puede lanzar
  IOException int
  dato = is.read();
} catch (IOException e) {
  System.out.println("Error leyendo del
  fichero");
} catch (FileNotFoundException e) {
```

Mostrará siempre "Error leyendo del fichero" sea cual sea el error que se produce.
Sería lo mismo que hacer:

```
try {
  // Abre el fichero en modo lectura
  // Puede lanzar FileNotFoundException
  FileInputStream is = new FileInputStream("mifichero");
  // Lee del fichero
  // Puede lanzar IOException
  int dato = is.read();
} catch (IOException e) {
    System.out.println("Error de fichero");
}
```

4.- Uso de objetos de las librerías estándar: String

Ahora que conocemos como crear y usar objetos vamos a ver como usaríamos una clase que ya existe.
Para ello vamos a emplear la clase `String`, que viene incluida en la librería estándar de Java. La librería estándar de Java son un conjunto de clases ya hechas que están disponibles para todas las aplicaciones Java, ya que se incluyen por defecto en todas las instalaciones. Otro tipo de librerías, que veremos más adelante, contienen clases que definen otros programadores y hay que especificar su uso de forma directa.

La clase `String` representa una cadena de caracteres. Los mensajes que ofrece están orientados, por tanto, a la manipulación de cadenas de caracteres y texto.

La clase está diseñada de tal forma que todos los objetos son *inmutables*, esto es, que una vez que un objeto de clase `String` contiene una cadena, ésta no se puede modificar. Una de las consecuencias de este diseño es que la mayoría de mensajes que vamos a ver no operan sobre el objeto que recibe el mensaje sino que se crea un nuevo objeto de clase String para almacenar el resultado, manteniendo el objeto original que recibió el mensaje sin cambios. Existe una clase alternativa para trabajar con cadenas de caracteres, la clase StringBuilder, que *si* permite la modificación de su contenido.

La documentación de la clase String la puedes encontrar en: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

4.1.- Creación de objetos String

Los objetos String se crean como el resto de objetos, usando new y los constructores. Hay una gran variedad de ellos pero los únicos que vamos a comentar porque son los más usados son:

- `String()`. Constructor por defecto. Crea una cadena vacía
- `String(String original)`. Crea una cadena con la misma

secuencia de caracteres que otra ya existente (`original`).

Asimismo, String se distingue del resto de clases del sistema en que permite un tercer método de construcción. En nuestro código podemos hacer:

```
String cadena = "Esto es un literal de cadena";
```

Esto crea un nuevo objeto de la clase String con el valor especificado en el literal. Nótese que la instrucción `new` no aparece por ningún lado.

4.2.- Lectura de cadenas desde el teclado

En bloques anteriores aprendimos la forma de leer datos de distintos tipos (int, long, double, etc.) desde teclado.
Para leer un dato de tipo String podemos utilizar la forma:

```
Scanner sc = new Scanner(System.in);

String dato = sc.nextLine();
```

Esto lee una cadena desde teclado, hasta que se pulsa la tecla `Enter` y la almacena en la cadena `dato`.

4.3.- Concatenación de cadenas

La concatenación de cadenas es una operación que toma 2 cadenas y devuelve otra formada por el contenido de la primera seguido por el contenido de la segunda. Por ejemplo, si concatenamos la cadena `"Hola"` con la cadena `"Adios"` obtendremos una nueva cadena con el contenido `"HolaAdios"`.

Para concatenar cadenas en Java no se emplea un mensaje sino el operador +, que ya hemos conocido como operador aritmético. Cuando uno de los operandos es una cadena se convierte en el operador concatenación de cadenas. Si el otro operando no es ya una cadena lo convierte a cadena. Veremos más sobre esto cuando veamos la herencia.

Por ejemplo, el código:

```
System.out.println("Hola " + "Caracola");
```

imprimiría por pantalla el texto `Hola Caracola`

4.4.- Mensajes de información

Los siguientes mensajes proporcionan información sobre la cadena:

- `int length()`. Devuelve el número de caracteres que componen la cadena. Puede valer cero si la cadena está vacía.

- `boolean isEmpty()`. Devuelve `true` si la cadena está vacía o `false` si no lo está. Es lo mismo que hacer `cadena.length() == 0` pero más sencillo y claro.

- `boolean isBlank()`. devuelve `true` si la cadena está vacía o sólo contiene caracteres blancos y `false` en caso contrario. Se consideran caracteres blancos los espacios, tabuladores o saltos de línea.

4.5.- Mensajes de comparación

Como hemos visto en secciones anteriores, para comparar dos objetos no basta con comparar las referencias, ya que esto sólo nos indica si las dos apuntan al mismo objeto, no si los valores contenidos en los dos objetos son "iguales" (el concepto de igualdad entre dos objetos lo veremos en bloques siguientes pero no es tan sencillo como comparar los valores de los atributos).
Por lo tanto, String ofrece varios métodos para comparar cadenas. Los métodos son, entre otros:

- `boolean equals(String otraCadena)`. Devuelve `true` si el contenido de las dos cadenas es igual o `false` en caso contrario. Es sensible a mayúsculas y minúsculas por lo que la comparación de `"Hola"` y `"hola"` devolverá false.

- `boolean equalsIgnoreCase(String otraCadena)`. Devuelve `true` si el contenido de las dos cadenas es igual o `false` en caso contrario. No es sensible a mayúsculas / minúsculas, por lo que la comparación de `"Hola"` y `"hola"` devolverá true.

- `int compareTo(String otraCadena)`. Devuelve un entero negativo (el valor no importa, sólo el signo) si esta cadena es anterior alfabéticamente a `otraCadena`, el valor 0 si las dos son iguales o un valor positivo (tampoco importa el valor) si esta cadena es alfabéticamente posterior a `otraCadena`. Versión sensible a mayúsculas / minúsculas.

- `int compareToIgnoreCase(String otraCadena)`. Igual que el anterior pero no es sensible a mayúsculas / minúsculas.

- `boolean contains(String otraCadena)`. Devuelve `true` si `otraCadena` está contenida dentro de la cadena que recibe el mensaje, incluyendo el caso en que ambas sean iguales. Por ejemplo (simulado), `"Hola".contains("la")` devuelve `true` ya que `"la"` forma parte de `"Hola"`.

- `boolean startsWith(String otraCadena)`. Devuelve `true` si esta cadena comienza por `otraCadena`. `false` en cualquier otro caso.

- `boolean endsWith(String otraCadena)`. Devuelve `true` si esta cadena termina por `otraCadena`. `false` en cualquier otro caso.

- `int indexOf(String otraCadena)`. Devuelve la posición, dentro de esta cadena, en la que se encuentra `otraCadena`. Si no se encuentra devuelve el valor `-1`.

- `int indexOf(String otraCadena, int posicion)`. Otra versión del mensaje que admite también un entero. Este indica la posición a partir de la que se va a buscar dentro de esta cadena.

- `int lastIndexOf(String otraCadena)`. Igual que los anteriores pero se comienza a buscar desde el final de la cadena hacia el inicio.

- `int lastIndexOf(String otraCadena, int posicion)`. Igual pero en lugar desde el final se comienza desde la posición dada.

4.6.- Mensajes de extracción

Estos mensajes se emplean para extraer partes de una cadena. Entre otros podemos encontrar:

- `char charAt(int indice)`. Devuelve el carácter situado en la posición `indice` de la cadena. indice puede valer desde 0 (para el primer carácter) hasta la longitud de la cadena
- 1 para el último. Si se da un indice incorrecto se produce un error.

- `String trim()`. Devuelve esta misma cadena eliminando los blancos al inicio y al final. Para ver qué caracteres son blancos consulta la descripción de `isBlank()`.

- `String substring(int posicion)`. Devuelve la sub-cadena de esta que comienza en la posición dada y sigue hasta el final de la cadena. La posicion debe seguir las mismas reglas que el indice de `charAt`.

- `String substring(int posicionInicial, int posicionFinal)`. Igual que el anterior pero en lugar de devolver hasta el final de la cadena devuelve sólo hasta la posición final dada, que debe ser mayor o igual que la inicial.

- `String toUpperCase()`. Convierte la cadena que recibe el mensaje de forma que todas las letras son mayúsculas. **OJO: Este método modifica la cadena que lo recibe. Devuelve referencia a la misma cadena.**

- `String toLowerCase()`. Exactamente igual que el anterior pero pasa todas las letras a minúsculas.

5. Enumerados

Hay veces que cuando estamos realizando un programa nos puede resultar útil que algunas variables pueda tomar una serie de valores que nos interese, como puede ser valores permitidos de días de la semana, colores de un semáforo…En estos casos nos interesa que estos valores sean únicos y que todos estén escritos de la misma forma. En Java esto podemos realizarlo con los enumerados, los cuales crean una serie de constantes que nos ayuda en esta tarea.
Podemos crear un enumerado de la siguiente forma:

```
enum Semaforo{
        ROJO, AMARILLO, VERDE
}
```

Y podemos crear una instancia de la forma:

```
Semaforo semaforo = Semaforo.verde;
```

Podemos observar Semaforo lo hemos puesto la primera letra en mayúsculas. **Esto es una convención de Java** porque realmente estamos creando un nuevo tipo de datos que tiene esos valores definidos. También podemos observar que hemos puesto los posibles valores en mayúsculas (ROJO, AMARILLO, VERDE) porque se consideran constantes. Loa enumerados en Java heredan de la clase java.lang.Enum.

Hay varios métodos útiles de enum:

- values() → devuelve todos los valores del enum
- valueOf(String) → convierte un texto en enum
- name() → nombre de la constante
- ordinal() → la posición que ocupa una constante. No es recomendable usarlo

Un ejemplo completo puede ser:

```java
public class Main {

    enum Semaforo {
        ROJO,
        AMARILLO,
        VERDE;
    }

    public static void main(String[] args) {

        Semaforo color = Semaforo.ROJO;
        System.out.println("Color actual: " + color);

        color = Semaforo.VERDE;
        System.out.println("Nuevo color: " + color);

        System.out.println("\nSe puede tilizar en un switch");
        switch (color) {
            case ROJO:
                System.out.println("El semáforo está en ROJO");
                break;
            case AMARILLO:
                System.out.println("El semáforo está en AMARILLO");
                break;
            case VERDE:
                System.out.println("El semáforo está en VERDE");
                break;
        }

        System.out.println("\nLos colores posibles del semáforo son:");
        for (Semaforo c : Semaforo.values()) {
            System.out.println(c.name());
        }

        System.out.println("\nEjemplo con valueOf():");
        String texto = "AMARILLO";
```

```
        // Convertir el String a enum usando valueOf
        Semaforo semaforo2 = Semaforo.valueOf(texto);

        System.out.println(semaforo2);

        try {
            Semaforo valor2 = Semaforo.AMARILLO;
            System.out.println("\nEl valor del semáforo es: " +valor2);
        } catch (IllegalArgumentException e) {
            System.out.println("Error capturado: " + e.getMessage());
        }

        // Generamos una excepción
        try {
               Semaforo semaforo3 = Semaforo.valueOf("amarillito");
               System.out.println(semaforo2);
        } catch (IllegalArgumentException e) {
            System.out.println("Error capturado: " + e.getMessage());
        }
    }
}
```

Los enumerados nos pueden permitir realizar muchas más acciones, podemos crear constructores y métodos y usarlos a nuestra conveniencia.

6. Tipo fecha en Java

Cuando Java comenzó a desarrollarse, Java dispuso de bibliotecas para tratar el tipo fecha, llamada Java ~~**java.util.Date y java.util.Calendar**~~. Sin embargo, actualmente no se utilizan porque hay una alternativa más fácil de utilizar, la biblioteca **java.time**.
El paquete java.time pertenece a la API de Java Time e incluye varias clases para trabajar con fechas, tiempos, duraciones, períodos y zonas horarias.
Hay que añadir, que los objetos tipo fecha en Java son inmutables.
Las principales clases de java.time son:

Clase	Descripción	Ejemplo de uso
LocalDate	Devuelve la fecha actual	LocalDate.now()
LocalTime	Devuelve la hora actual	LocalTime.now()
LocalDateTime	Fecha y hora actual	LocalDateTime.now()
ZonedDateTime	Fecha y hora con zona horaria.	ZonedDateTime.now()

Instant	Momento exacto en el tiempo en UTC.	Instant.now()
Duration	Intervalo de tiempo en horas, minutos, segundos.	Duration.between(t1, t2)
Period	Diferencia en años, meses y días.	Period.between(d1, d2)
ChronoUnit	Permite medir la diferencia entre fechas en distintas unidades.	ChronoUnit.DAYS.between(d1, d2)
ZoneId	Representa una zona horaria (Ej: America/Mexico_City).	ZoneId.of("Europe/Paris")
DateTimeFormatter	Formatea fechas y horas.	DateTimeFormatter.ofPattern("dd/MM/yyyy")

Un ejemplo puede ser:

```
package fecha;
import java.time.*;
import java.time.format.DateTimeFormatter;
public class Fecha {
	public static void main(String[] args) {
		// Obtener la hora actual en UTC
		Instant ahoraUTC = Instant.now();
		System.out.println("Hora actual en UTC: " + ahoraUTC);

		// Convertir UTC a diferentes zonas horarias
		ZonedDateTime horaMadrid = ahoraUTC.atZone(ZoneId.of("Europe/Madrid"));
		ZonedDateTime horaNewYork = ahoraUTC.atZone(ZoneId.of("America/New_York"));
		ZonedDateTime horaTokio = ahoraUTC.atZone(ZoneId.of("Asia/Tokyo"));
		System.out.println("Hora en Madrid: " + horaMadrid);
		System.out.println("Hora en Nueva York: " + horaNewYork);
		System.out.println("Hora en Tokio: " + horaTokio);

		// Obtener solo la fecha y la hora local del sistema
		LocalDate fechaActual = LocalDate.now();
		LocalTime horaActual = LocalTime.now();
		LocalDateTime fechaHoraActual = LocalDateTime.now();
		System.out.println("Fecha actual: " + fechaActual);
		System.out.println("Hora actual: " + horaActual);
		System.out.println("Fecha y hora actual: " + fechaHoraActual);

		// Calcular la diferencia entre dos fechas con Period
		LocalDate fechaInicio = LocalDate.of(2020, 5, 10);
		LocalDate fechaFin = LocalDate.of(2025, 2, 17);
		Period periodo = Period.between(fechaInicio, fechaFin);
		System.out.println("Diferencia: " + periodo.getYears() + " años, " + periodo.getMonths() + "
```

```
meses, "
                                        + periodo.getDays() + " días.");

                    // Calcular la diferencia entre dos tiempos con Duration
                    LocalTime hora1 = LocalTime.of(14, 30);
                    LocalTime hora2 = LocalTime.of(18, 45);
                    Duration duracion = Duration.between(hora1, hora2);
                    System.out.println(
                                        "Diferencia de tiempo: " + duracion.toHours() + " horas, " +
duracion.toMinutesPart() + " minutos.");

                    // Formatear una fecha y hora personalizada
                    DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
                    String fechaFormateada = fechaHoraActual.format(formato);
                    System.out.println("Fecha y hora formateada: " + fechaFormateada);
          }
}
```

siendo la salida algo como esto:

```
Hora actual en UTC: 2025-03-02T22:56:09.905906800Z
Hora en Madrid: 2025-03-02T23:56:09.905906800+01:00[Europe/Madrid]
Hora en Nueva York: 2025-03-02T17:56:09.905906800-05:00[America/New_York]
Hora en Tokio: 2025-03-03T07:56:09.905906800+09:00[Asia/Tokyo]
Fecha actual: 2025-03-02
Hora actual: 23:56:09.949351
Fecha y hora actual: 2025-03-02T23:56:09.949351
Diferencia: 4 años, 9 meses, 7 días.
Diferencia de tiempo: 4 horas, 15 minutos.
Fecha y hora formateada: 02/03/2025 23:56:09
```

Hay tres métodos que podemos utilizar con estas clases:

- **now()**, lo hemos visto en el ejemplo anterior, crea una instancia con la fecha y/o la hora actual.4
- **of(),** permite crear una fecha y/o hora específica en lugar de tomar la actual.
- **with(),** modifica una parte específica de un objeto de fecha/hora, devolviendo una nueva instancia

Un ejemplo con of() puede ser el siguiente:

```
package fecha;
          import java.time.*;
          public class Fecha2 {
                    public static void main(String[] args) {
                              LocalDate fecha = LocalDate.of(2025, 3, 2);
                              LocalTime hora = LocalTime.of(22, 56, 9);
                              LocalDateTime fechaHora = LocalDateTime.of(2025, 3, 2, 22, 56, 9);
                              System.out.println("Fecha específica: " + fecha);
                              System.out.println("Hora específica: " + hora);
                              System.out.println("Fecha y hora específica: " + fechaHora);
                    }
          }
```

siendo la salida:

```
Fecha específica: 2025-03-02
Hora específica: 22:56:09
Fecha y hora específica: 2025-03-02T22:56:09
```

Con with() podemos cambiar la fecha como en el ejemplo:

```
package fecha;
import java.time.*;
public class Fecha3 {


	public static void main(String[] args) {
		LocalDate fechaOriginal = LocalDate.of(2025, 3, 2);
		LocalDate nuevaFecha = fechaOriginal.withYear(2030).withMonth(12).withDayOfMonth(25);
		System.out.println("Fecha original: " + fechaOriginal);
		System.out.println("Nueva fecha con modificaciones: " + nuevaFecha);
	}
}
```

con salida:

```
Fecha original: 2025-03-02
Nueva fecha con modificaciones: 2030-12-25
```

Aunque también la hora, los minutos y los segundos:

```
package fecha;
import java.time.LocalTime;
public class Fecha4 {
	public static void main(String[] args) {
		LocalTime horaOriginal = LocalTime.of(22, 56, 9);
		LocalTime nuevaHora = horaOriginal.withHour(10).withMinute(30).withSecond(50);
		System.out.println("Hora original: " + horaOriginal);
		System.out.println("Nueva hora modificada: " + nuevaHora);
	}
}
```

siendo la salida:

```
Hora original: 22:56:09
Nueva hora modificada: 10:30:50
```

Partes de una fecha o una hora

En Java, una fecha u hora se compone de varias partes. Por ejemplo, en la clase LocalDate, puedes obtener el año, mes y día de una fecha específica usando métodos como getYear(), getMonthValue() y getDayofMonth().

LocalDate date = LocalDate.now();

int year = date.getYear();

```
int month = date.getMonthValue();

int day = date.getDayOfMonth();

System.out.println("Año: " + year + ", Mes: " + month + ", Día: " + day);
```

De igual manera se puede hacer con las partes de una hora, como hora, minuto y segundo. En este caso, se usan métodos como getHour(), getMinute() y getSecond().

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

int hour = time.getHour();

int minute = time.getMinute();

System.out.println("Hora: " + hour + ", Minuto: " + minute);
```

Tiempo transcurrido entre fechas y horas

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

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

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

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

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

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

```
package fecha;
import java.time.*;
import java.time.temporal.TemporalAdjusters;
public class Fecha4 {
```

```
        public static void main(String[] args) {
                LocalDate hoy = LocalDate.now();
                LocalDate finDeAnio = hoy.with(TemporalAdjusters.lastDayOfYear());
                Period hastaFinDeAnio = hoy.until(finDeAnio);
                int meses = hastaFinDeAnio.getMonths();
                int dias = hastaFinDeAnio.getDays();
                System.out.println("Faltan " + meses + " meses y " + dias + " días hasta final de año.");
        }
}
```

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

```
package fecha;

import java.time.LocalDate;
import java.time.Period;

public class Fecha {
        public static void main(String[] args) {
                LocalDate fechaInicio = LocalDate.of(2023, 10, 15);
                LocalDate fechaFin = LocalDate.of(2024, 12, 20);

                Period diferencia = Period.between(fechaInicio, fechaFin);

                System.out.println("La diferencia es: " + diferencia.getYears() + " años,
" + diferencia.getMonths() + " meses y " + diferencia.getDays() + " días.");
                // Salida: La diferencia es: 1 años, 2 meses y 5 días.
        }
}
```

11.- Referencias

- Explicación alternativa sobre clases en Java (https://codesitio.com/recursos-utiles-para-tu- web-o-blog/cursos/curso-de-java-clases-atributos-modificadores-objetos-y-metodos/)