Programación

Bloque 04 - Desarrollo de Clases

Índice

1.- Introducción

Hasta este momento hemos utilizado clases hechas por otros, ya sea en forma de librerías o bien en forma de clases predefinidas que se incluyen con Java.

En este bloque vamos a introducir cómo crear nuestras propias clases, tanto para nuestras aplicaciones como para crear librerías que puedan ser reutilizadas en el futuro, tanto por nosotros mismos como por otros.

2.- Repaso de conceptos básicos

Aunque ya los introdujimos en el bloque 2 vamos a repasar conceptos que o bien necesitamos para desarrollar este tema o bien vamos a desarrollar directamente en este tema.

2.1.- Objeto

Como definimos en el bloque 2, un objeto es una representación en el ordenador de algo que puede existir en el mundo real.

Un objeto tiene como propiedades la identidad, el estado y el 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 o atributos 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.

2.2.- Clase

Los objetos de un programa se definen mediante *clases*. Ya que es imposible el programar todos los posibles objetos que vayamos a tener en un sistema, a no ser que sean unos pocos, la programación se realiza por clases.

Una clase es un grupo de objetos que comparten características comunes. Específicamente tienen el mismo conjunto de atributos (aunque los valores normalmente diferirán 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.

Por lo tanto, a la hora de desarrollar una aplicación orientada a objetos, el programador programa clases, aunque en el momento de funcionar se creen objetos pertenecientes a dichas clases, que son los que realmente definen el comportamiento dinámico de la aplicación.

3.- Estructura y miembros de una clase

En Java, una clase se declara usando la palabra clave `class`, seguida del nombre de la clase, seguida de un bloque en el que se definen los componentes o ***miembros*** de una clase.

Estos miembros pueden ser:

- **Atributos.** Son las propiedades que tienen los objetos. Cada atributo tendrá un tipo determinado.
- **Métodos.** Son trozos de código que sirven para responder a los mensajes que reciben los objetos de la clase.
- **Constructores.** Son métodos especiales que se invocan cuando se construyen nuevos objetos de una clase y sirven para inicializarlos.

Además hay que añadir que tanto los atributos como los métodos pueden ser de instancia, esto es, se aplican a instancias de la clase como estáticos, o sea que se aplican a la clase en si.

Ejemplo:

```
public class NombreClase {
  // Atributos
  .....
  // Constructores
  .....
  // Métodos
  .....
}
```

En las siguientes secciones describiremos la forma de crear cada uno de los tipos de miembro, así como sus peculiaridades. Sin embargo, a continuación vamos a describir una característica fundamental, tanto de Java como de la programación orientada a objetos que es la ***visibilidad*** y que se aplica a todos los miembros de cualquier clase. En el ejemplo, la clase tiene una visibilidad pública porque usa el modificador `public`, que veremos más adelante.

3.1.- 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.

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. El complemento del interfaz de una clase es la *implementación* de la misma, que define el funcionamiento interno de la clase.

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.

Esto obviamente no siempre es posible ya que a veces se requieren hacer cambios a una clase que requieren modificar el interfaz pero hay que intentar evitar esto todo lo que se pueda.

Por lo tanto, Java proporciona un mecanismo, llamado ***control de visibilidad*** que permite especificar, para cada miembro de una clase, si este 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 relativamente simple. Cada miembro puede indicar, mediante una palabra clave, su visibilidad. Las palabras clave y su significado son las siguientes:

- `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 (más tarde veremos más sobre herencia).
- `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.

Además existe un tipo adicional de visibilidad, la visibilidad a nivel de paquete, que es la que tiene un miembro si no indica ninguna de las tres anteriores. En la visibilidad a nivel de paquete, el miembro es accesible a la clase en que se define y por todas las clases contenidas en el mismo paquete que aquella.

El modificador de visibilidad debe ser lo primero que debe aparecer en la declaración del miembro.

4.- Atributos

La declaración de un atributo es bastante sencilla. Se declara igual que una variable pero se hace como parte del cuerpo de una clase. La forma general de declaración de un atributo es:
`[visibilidad] [declaracion_estatica] tipo nombre [inicializacion];`
donde:

- `visibilidad` es el indicador de visibilidad, ya visto anteriormente. Puede ser uno de `public`, `private`, `protected` o ninguno (visibilidad a nivel de paquete).
- `declaracion_estatica`. Indica mediante la palabra clave `static` que el atributo es de clase en lugar de instancia. Si es un atributo de instancia no lleva la palabra (`static`).
- `tipo`. Tipo del atributo. Igual que para la declaración de una variable.
- `nombre`. Nombre del atributo. Debe ser único para la clase, esto es, no puede haber dos atributos con el mismo nombre.
- `inicializacion`. Valor inicial del atributo. Lo veremos más tarde cuando hablemos de los constructores.

Ejemplos:

```
public int edad;
private static String nombre;
protected double porcentaje;
static long cantidad;
```

El primer ejemplo define un atributo llamado `edad`, de tipo `int` y visibilidad pública y perteneciente a la instancia.
El segundo es un atributo llamado `nombre`, de tipo referencia a `String`, visibilidad privada y perteneciente a la clase.
El tercero es un atributo llamado `porcentaje`, de tipo `double`, visibilidad protegida y perteneciente a la instancia.
Y por último el cuarto es un atributo llamado `cantidad`, de tipo `long`, visibilidad a nivel de paquete y perteneciente a la clase.
Como ya se comentó en bloques anteriores, hay que intentar evitar siempre que sea posible la utilización de atributos con visibilidad pública. La mejor opción es utilizar siempre visibilidad privada y levantarla a protegida o de paquete sólo cuando sea necesario.

5.- 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, que debe coincidir con el mensaje, esto es, el método `calcular` procesa el mensaje `calcular`.
- `lista_parametros` es la lista de parámetros del método. Indica el número de parámetros que se reciben por el ḿetodo 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. Cada parámetro tiene la siguiente sintaxis:
  tipo nombre
  donde:
  - `tipo` es el tipo del parámetro. Puede ser un tipo primitivo (int, long, double, etc.) o una referencia a un objeto (String, por ejemplo). El valor que se proporcione para este parámetro deberá poder ser convertido a este tipo.
  - `nombre` es el nombre del parámetro y es aquel por el que se podrá referenciar dentro del cuerpo del método.

A la línea inicial de la declaración de un método que contiene toda esta información (nombre, tipo visibilidad, parámetro, etc.) se le denomina en la jerga del oficio como ***cabecera***, ***firma*** o ***signatura*** (signature en inglés) del método.

Ejemplos:

A continuación tenemos varios ejemplos de declaración de métodos. Vamos a obviar el cuerpo de los mismos por el momento ya que aún tenemos que discutir algunas cosas, especialmente el paso de parámetros y la devolución de valores de retorno.

```
public void iniciar() {
 ....
}


private static int longitudCadena(String cadena) {
  .....
}


protected String getDNI(String nombre, String apellidos, int edad) {
   ......
}
```

La primera declaración define un método llamado `iniciar`, de visibilidad pública, de instancia (no tiene `static`), no tiene parámetros y no devuelve nada (`void`).

La segunda define un método llamado `longitudCadena`, de visibilidad privada, de clase (estático), tiene un parámetro, `cadena`, de tipo `String` y devuelve un valor entero (`int`).

La última define un método llamado `getDNI`, de visibilidad protegida, de instancia (no es estático), tiene tres parámetros: `nombre` de tipo `String`, `apellidos` de tipo `String` y `edad` de tipo `int`. Devuelve una referencia a un objeto de tipo `String`.

5.1.- Mecanismo de paso de parámetros

El mecanismo de paso de parámetros permite transmitir información entre el objeto que envía el mensaje y el objeto que lo recibe en el momento en que se realiza el envío. Esto, como ya vimos en bloques anteriores, permite enviar información necesaria para realizar la acción que dispara el mensaje. Para que se produzca el paso de parámetros se requiere que el método declare una lista de parámetros y que el objeto que realiza la llamada proporcione un valor para cada parámetro de la lista.

Por ejemplo, supongamos que tenemos una clase, llamada `Receptor`, que define un método mensaje1, con la siguiente firma:

```
public void mensaje1(int parametro1, long p2, String valor3) {
  .....
}
```

Esta declaración describe este método como público, no devuelve nada, es de instancia (no es estático) y recibe tres parámetros, que son, en orden:

- `parametro1`, de tipo `int`
- `p2`, de tipo `long`
- `valor3`, de tipo `String`

Cuando se quiere enviar el mensaje correspondiente a este método desde otro objeto, que llamaremos por ejemplo `Emisor`, hay que proporcionar tres valores para los tres parámetros y cada valor debe ser compatible con el tipo del parámetro correspondiente. Los valores y los parámetros

se emparejan según el orden en que aparezcan tanto en la declaración del método como en la lista de valores a la hora de enviar el mensaje, luego lo siguiente sería correcto:

```
public class Emisor {
  .......
  public static void main(String[] args) {
    ........
    long variable = 2;
    Receptor receptor = new Receptor();
    receptor.mensaje1(2, 2 * variable, "cadena");
    .....
  }
}
```

Como se ve, en la séptima línea se produce el envío del mensaje desde el emisor al receptor. Dado que la lista de parámetros del método mensaje1 tiene tres parámetros, de tipos `int`, `long` y `String`, hay que proporcionar tres valores de estos tipos y en ese orden (`2`, `2 * variable` = 4 y `"cadena"`).

Es importante hacer notar que los tipos de los valores que se envían deben ser compatibles con los tipos de los parámetros que reciben. Por ejemplo, en el primer caso, al ser de tipo `int`, debe recibir un valor de tipo `int`. Si se intentara pasar un valor de tipo `long` (`2L`, por ejemplo), el código no compilaría. En el caso del segundo parámetro, dado que es de tipo `long`, se podría dar un valor `long` (lo esperado) pero también `int`, ya que Java puede convertir automáticamente de `int` a `long`. En el caso del tercer parámetro hay que pasar una referencia a un `String`. En este caso se proporciona un literal de cadena que Java convierte automáticamente a un objeto de tipo `String`.

Este envío de mensaje "transfiere" los tres valores (2, 4 y "cadena") desde el emisor hacia el receptor. Lo siguiente que hay que determinar es: ¿Cómo se reciben estos valores en el lado receptor?.

La respuesta es sencilla. En la definición del método se ha declarado la lista de parámetros indicando el tipo y el nombre de cada uno. El tipo ya hemos visto como se utiliza pero, ¿para qué sirven los nombres? Los nombres se emplean para recibir los valores de los parámetros.

Cuando se entra en el bloque del método, Java ha definido ***automáticamente*** tres variables, con los nombres y tipos de los parámetros y en ellas se colocan, también automáticamente, los valores enviados por el emisor. De esta forma, efectivamente la información se ha transferido desde el emisor hacia el cuerpo del método.

En el cuerpo del método se pueden usar estas variables como si se hubieran definido en el mismo cuerpo.

Es importante reseñar un par de aspectos importantes respecto a este paso de parámetros. El primero es que los parámetros se ***copian*** desde el emisor hacia el receptor. Esto significa que en el caso de los tipos primitivos, los valores se copian desde el emisor hacia las variables que los reciben. En el caso de que los parámetros sean objetos, se copian ***las referencias*** a los objetos, lo que significa que el mismo objeto que se accede desde el emisor, se accede desde el receptor, ya que la referencia es la misma.

El segundo aspecto a reseñar es que los parámetros ***se pueden modificar***, esto es, las variables que representan a los parámetros se pueden cambiar dentro del cuerpo del método. ¿En qué afecta esto

al paso de parámetros? La respuesta es: en nada, dado que los valores modificados se perderán en cuanto se acabe el bloque en que se definen, en este caso, el bloque del método.

Un último detalle: El cuerpo de un método define su propio ámbito de existencia de las variables (incluyendo en estas a los parámetros). Esto significa en la práctica que dentro del cuerpo de un método podemos nombrar a las variables como queramos dado que sus nombres no "chocarán" nunca con otras variables definidas en otros métodos.

Esto nos permite enfocar el desarrollo del cuerpo de un método como si fuera un "miniprograma" ya que crear las variables que queramos, con los nombres que queramos y centrarnos en realizar la tarea que sea la que debe realizar el método sin tener que preocuparnos por interacciones con otras partes del programa. La única interacción se produce mediante el paso de parámetros, que ya se ha descrito, el valor de retorno, que se describirá más adelante y con los atributos (de la clase o de la instancia), que se describirán en la siguiente sección.

5.2.- 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

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

Cuando se invoca (envía un mensaje) a un método de instancia, el objeto sobre el que se ha enviado el mensaje (recordar que el envío de un mensaje se realiza mediante la sintaxis `objeto.metodo`) 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" (más sobre inicialización y constructores más adelante).

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 asi 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 "mágica" 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`.

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

Los métodos de clase (static) 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 linea 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.

5.3.- 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.

5.4.- 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.

5.5.- Sobrecarga de métodos

En Java es posible escribir más de un método para el mismo mensaje.

Los métodos deben tener todos el mismo nombre, modificador (estático o no) y tipo del valor de retorno, pero debe variar la lista de parámetros. Concretamente, la lista de parámetros, ***eliminando los nombres de los parámetros***, debe ser distinta para todos los métodos con el mismo nombre.

El siguiente ejemplo sería válido:

```
public class UtilidadesParalelogramos {
  public static double area(double lado) {
    return lado * lado;
  }

  public static double area(double ladoHorizontal, double ladoVertical) {
    return ladoHorizontal * ladoVertical;
  }
}
```

Como se puede ver hay dos métodos para el mensaje `area`. El primero recibe como parámetros un `double` y el segundo dos `doubles`.

Otro ejemplo válido sería:

```
public class UtilidadesCalculo {
  public static double potencia(int base, double exponente) {
    return Math.pow(base, exponente);
```

```
  }

  public static double potencia(double base, double exponente) {
    return Math.pow(base, exponente);
  }
}
```

En este caso, a pesar de que los dos métodos reciben dos parámetros, los tipos no son iguales para el primer parámetro.

Sin embargo, el siguiente ejemplo NO es válido:

```
public class UtilidadesCirculo {
  public static double area(double radio) {
    return Math.PI * radio * radio;
  }

  public static double area(double diametro) {
    return Math.PI * (diametro / 2) * (diametro / 2);
  }
}
```

En este caso el compilador daría error porque las listas de parámetros tienen el mismo número y tipo. El nombre de los parámetros es distinto pero dado que no se tiene en cuenta, es irrelevante para distinguir entre los dos métodos.
Otro ejemplo NO válido:

```
public class UtilidadesMatematicas {
  public static double potencia(double base, double exponente) {
    return Math.pow(base, exponente);
  }

  public static int potencia(int base, int exponente) {
    return Math.pow(base, exponente);
  }
}
```

Este caso tampoco compila porque los tipos de los valores de retorno son distintos y deben ser iguales para todos los métodos con el mismo nombre.
Queda una pregunta en el aire: Si todos los métodos para un mismo mensaje tienen el mismo nombre, ¿cómo sabe Java cual de los métodos ejecutar para responder a un envío de mensaje? La respuesta, seguro que has adivinado, es examinando el número y tipo de los valores que se van a pasar como parámetros del mensaje. Java examina cuantos valores se pasan, en qué orden y de qué tipo son cada uno y busca entre los metodos uno que coincida con esta lista. Si lo encuentra, este es el método seleccionado y el que se ejecuta para responder al mensaje. Si no lo encuentra, intentará promocionar valores de forma implícita para que encajen con algún método.

5.6.- 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 ahi 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 setCasado(boolean casado)
```

6.- 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.

6.1.- 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.

6.2.- 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.

6.3.- 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 si.

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.

6.4.- Inicialización estática de atributos

Los atributos también se pueden utilizar de la manera a la que estamos aconstumbrados 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.

6.5.- 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`

7.- Documentación de clases

Hasta este momento hemos usado los comentarios para describir partes o bloques de nuestro programa. En esta sección vamos a extender la documentación para incluir las clases que creemos y sus partes.

Para ello vamos a usar el formato JavaDoc. El formato JavaDoc se introdujo casi con la primera versión de java y es el formato de facto para la documentación.

Un comentario JavaDoc comienza por /** (dos asteriscos) y termina con */ (como los comentarios mutilínea normales).

Dentro de ellos cada línea puede comenzar por espacios, seguidos de * y del contenido real.

Ejemplo:

```
/**
 * Este es un comentario JavaDoc
 */
```

Los asteriscos al inicio de la línea no forman parte de la documentación.

Dentro de un comentario JavaDoc se puede incluir HTML pero nosotros no vamos a emplearlo excepto para alguna negrita o curisiva.

Además del texto, se puede emplear etiquetas (tags) que proporcionan información, la cual se recopila y con las que se hacen tablas o referencias cruzadas. Las etiquetas comienzan por el carácter arroba (@). Entre otras tenemos las siguientes etiquetas:

Tag	Descripción	Uso
@author	Nombre del desarrollador.	nombre_autor
@version	Versión del método o clase.	versión
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	nombre_parametro descripción
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".	descripción
@throws	Excepción lanzada por el método, posee un sinónimo de nombre @exception	nombre_clase descripción

El comentario se coloca justo antes de lo que se va a comentar: clase, atributo, método.

Para los atributos, ya que normalmente son privados basta con un comentario "normal", pero la clase y los métodos si deberían ir comentados, incluyendo los privados o protegidos.

Ejemplo de documentación de clase

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

```
public class Persona {

}
```

Para los métodos podría ser algo como lo siguiente:

```
  /**
   * 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 Manuel Montoro
   */
  public double calculaMedia3(double valor1, double valor2, double valor3) {
    return (valor1 + valor2 + valor3) / 3.0;
  }
```

Una vez que tenemos todas nuestras clases comentadas, la utilidad javadoc, desde la línea de comandos nos generará automáticamente archivos HTML navegables con la documentación de nuestras clases.

Si usamos Eclipse, este también puede hacerlo de forma muy sencilla y sin usar la línea de comandos.

8.- Creación y lanzamiento de excepciones

Un tipo especial de clase, que hemos visto a medias hasta ahora son las excepciones. Las excepciones son instancias de clases especiales (las clases de excepcion) y que contienen la información relativa a una excepción cuando esta ocurre. Hay que recordar que una de las informaciones que contiene una excepción es la clase a la que pertenece en sí, que ya da una indicación del tipo de error que ha ocurrido.

Java proporciona un gran número de clases de excepciones ya definidas en su vasta librería de clases predefinidas pero no es necesario que nos limitemos a estas ya que nosotros podemos crear nuestras propias excepciones, así como provocarlas (acción a lo que nos referiremos a partir de ahora como "lanzar" la excepción).

8.1.- Creación de excepciones

El primer paso es crear nuestra propia excepción.

Antes de crearla tenemos que saber que Java soporta dos tipos de excepciones, las excepciones chequeadas y las excepciones no chequeadas (mas sobre esto más tarde).

Para crear una excepción chequeada debemos crear una nueva clase con el nombre que queramos (aunque por convención el nombre de la clase debería terminar con la palabra `Exception`) y hacerla que herede de la clase `Exception`. No vamos a entrar en el funcionamiento de la herencia por ahora pero se puede hacer fácilmente desde Eclipse y escribiendo haciendo la siguiente declaración de la clase:

```
public class MiChequeadaException extends Exception {
  ....
}
```

En principio no es necesario incluir nada en el cuerpo de la clase. En bloque siguientes veremos como customizarla, pero por ahora se puede quedar asi.

Crear una excepción **no** chequeada es igual de fácil haciéndola que herede de `RuntimeException`, en lugar de `Exception`.

```
public class MiNoChequeadaException extends RuntimeException {
  ....
}
```

Una vez hecho esto, ya tenemos nuestra excepción, chequeada o no, creada y podemos pasar al siguiente paso, usarla.

8.2.- Lanzamiento de excepciones

Cuando, en la implementación de nuestros métodos nos encontremos con que llegamos a una condición excepcional que nos impide continuar, es el momento de lanzar nuestra excepción. Esto es algo muy sencillo de realizar siguiendo los siguientes pasos:

- Crea un objeto del objeto excepción que quieras lanzar, por ejemplo, `MiChequeadaException exception = new MiChequeadaException()`, creará una excepción chequeada.

- Lanzarla. Esto se hace mediante la instrucción `throw objeto_excepcion`. Esto provoca que la ejecución del método en cuestión se termine inmediatamente. Las posibles instrucciones que continuaran detrás de `throw` se descartan. En este sentido funciona en cierto modo igual que `return`.

Si se quiere se puede hacer las dos cosas en el mismo paso:

```
throw new MiChequeadaException();
```

Esta linea crea la excepción y la lanza. Todo en uno. También es la forma más normal de hacerlo, por otra parte.

A continuación comienza el proceso que vamos a denominar "búsqueda de un manejador con relanzamiento". El proceso es el siguiente:

- Si el envío del mensaje al método que ha lanzado la excepción está dentro de un bloque try..catch con la excepción correspondiente (MiChequeadaException en el caso del ejemplo), se usa ese manejador y se da por finalizada el proceso de la excepción.
- Si el envío en cuestión no estaba dentro de un bloque try...catch, la excepción se "relanza", en este caso al envío del mensaje que inició la ejecución de este segundo método, terminando inmediatamente la ejecución de este método.
- El proceso se repite hasta que se llega a main, que es el punto de inicio en última instacia de la aplicación. Si en main, tampoco hay ningún manejador, Java realiza el manejo por defecto que es terminar la aplicación y mostrar una traza. Esta traza contiene una lista de los métodos por los que se ha ido lanzando y relanzando la excepción, desde el método original hasta main, incluyendo la línea del programa en que se ha lanzado (o relanzado) la excepción.

Un ejemplo

Supongamos la clase `Persona`:

```
public class Persona {

  public Persona() {
  }

  public void metodo1() {
    System.out.println("Metodo 1");
    metodo2();
  }

  public void metodo1Robusto() {
    System.out.println("Metodo 1 Robusto");
    try {
      metodo2();
    } catch (MiNoChequeadaException e) {
      System.out.println("Excepcion manejada!!!!");
    }
  }

  private void metodo2() {
    System.out.println("Metodo 2");
    metodo3();
```

```
  }

  private void metodo3() {
    System.out.println("Metodo 3");
    throw new MiNoChequeadaException();
  }
}
```

Y la clase PruebaPersona, que tiene main:

```
public class PruebaPersona {

  public static void main(String[] args) {
    Persona persona = new Persona();
    persona.metodo1();
    System.out.println("No deberiamos llegar a esta linea");
  }
}
```

Cuando se ejecuta esta aplicación, se crea un objeto persona y se le envía el mensaje metodo1. Este, a su vez, envía el mensaje metodo2 al mismo objeto y éste mensaje envía a su vez el mensaje metodo3. En este momento main ha "llamado" a metodo1, metodo1 a metodo2 y metodo2 a metodo3.

Ahora, aparte de los mensajes, en metodo3 se lanza una excepción de la clase MiNoChequeadaException. Esto provoca que la ejecución se termine (aunque es la última instrucción de todas formas) y se lance la excepción hacia el emisor del mensaje. En este caso el emisor está en el método metodo2.

En metodo2 no hay tratamiento de excepciones (no bloque try...catch). Por lo tanto la excepción se "relanza" hacia el emisor del mensaje (en este caso hacia metodo2), que es metodo1.

En metodo1 se repite el patrón del método anterior (no hay try...catch), por lo que la excepción se vuelve a relanzar. En este caso al emisor del mensaje, que es main.

main tampoco tiene try..catch por lo que intenta relanzar la excepción pero como no hay emisor del mensaje, ya que main lo invoca el sistema Java en persona, es éste el que recibe la excepción. La aplicación acaba y se imprime la traza. En este caso lo que sale por pantalla sería, mas o menos:

```
Metodo 1

Metodo 2

Metodo 3

Exception in thread "main" MiNoChequeadaException

     at Persona.metodo3(Persona.java:29)

     at Persona.metodo2(Persona.java:24)

     at Persona.metodo1(Persona.java:10)

     at PruebaPersona.main(PruebaPersona.java:7)
```

Como puedes ver, se ven los mensajes y la traza, que comienza por "`Exception...`"

De arriba abajo puedes ver el punto donde se originó inicialmente la excepción (El metodo `metodo3` de la clase `Persona`, localizado en el archivo `Persona.java`, línea 29), de ahi se

lanzó al método `metodo2` de la clase persona localizado en la línea 24 del mismo archivo. De ahi pasó al método `metodo1` de la misma clase, localizado en la línea 10 y por último pasó al método `main` de la clase `PruebaPersona` (la clase de entrada de la aplicación) en la línea 7 de dicho archivo. De ahi se supone que pasó al sistema Java, que no nos revela nada sobre si mismo.

Si cambiáramos `PruebaPersona` de la siguiente manera:

```
public class PruebaPersona {

  public static void main(String[] args) {
    Persona persona = new Persona();
    persona.metodo1Robusto();
    System.out.println("No deberiamos llegar a esta linea");
  }
}
```

Ahora la cosa cambia. Al lanzarse la excepción en `metodo3`, ésta se pasa a `metodo2`. Este no dispone de bloque `try..catch`, por lo que se relanza a `metodo1Robusto`. Este **si** que dispone de `try..catch` adecuado por lo que la excepción se trata y no se relanza más.

8.3.- Declaración de excepciones

Aún no hemos tratado la diferencia entre las excepciones chequeadas y las que no lo son. Llegó el momento de comentar la diferencia.

La diferencia fundamental es que, con las opciones *chequeadas*, Java hace una comprobación en cáda método que pueda recibir una excepción de este tipo (porque se pueda lanzar desde algún mensaje que se envía desde dicho método) que el método en cuestión:

- O bien captura y maneja la excepción usando try..catch
- O bien declara en la cabecera del método que puede lanzar (o relanzar, en este caso) la excepción en cuestión.

La declaracion en la cabecera del método se hace incluyendo entre la lista de parámetros y la llave de inicio del bloque la palabra clave `throws` (ojo, con s al final), seguida de una lista, separada por comas, de las excepciones que pueden ser lanzadas desde el método. Java usa esta lista para, a su vez, chequear los métodos que puedan enviar mensajes al método en cuestión y asi sucesivamente hasta cubrir todas las posibilidades.

Esto tiene la ventaja de que se asegura que al desarrollar las clases el programador sea consciente de las excepciones que puede recibir por parte de los mensajes que envíe desde esas clases y deba hacer algo sobre ellos (tratarlas o decir que las deja pasar). Lo que no puede hacer es ignorarlas.

Este era (y es aún) un problema en lenguajes antiguos que o bien no disponían de excepciones o bien usaban el método de valores de retorno especiales para indicar exito o fracaso en el mensaje. Lo normal es que muchos programadores no fueran conscientes de estos valores de retorno y los ignoraran con lo cual las aplicaciones no respondían de forma ordenada a problemas que pudieran ocurrir (que es para lo que están las excepciones).

La desventaja es precisamente la misma. Exige al programador un tedioso chequeo / creación de manejadores, que ralentiza el trabajo y más teniendo en cuenta que la mayoría de las veces

simplemente se deja la excepción relanzar, con lo cual las declaraciones de los métodos se pueden hacer muy largas.

Además crea un problema de dependencias ya que cualquier clase depende de todas las excepciones que pueda recibir, haga lo que haga con ellas.

Las excepciones no chequeadas, sin embargo, se saltan este chequeo, por lo que el programador es libre de tratarlas o ignorarlas.

Sobre el uso de excepciones chequedas frente a no chequeadas hay un debate en el seno de la comunidad, con bandos enquistados en una u otra postura, aunque la que parece ganar peso últimamente es la de las excepciones no chequeadas.

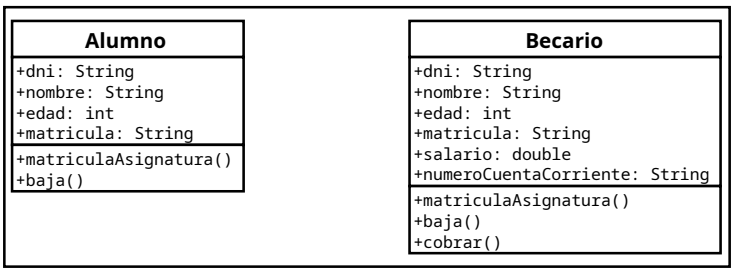
En este curso seguiremos esta escuela y usaremos preferentemente excepciones no chequeadas.

9.- 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.

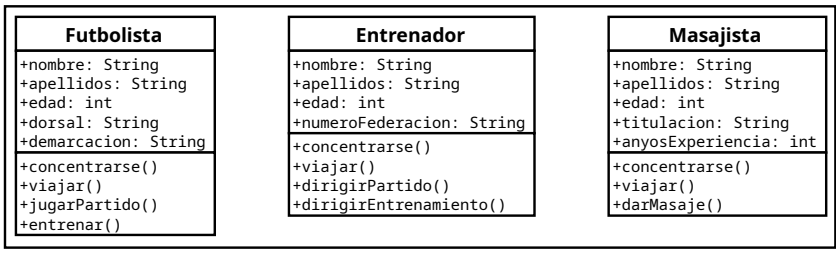
Si examinamos dichas clases en muchas ocasiones, no siempre, percibiremos que la similaridad proviene de que las dos clases realmente están relacionadas en el sentido de que, o bien una de las dos es un tipo especial de la otra, o bien ambas son tipos especiales de una tercera.

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), asi 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 estas similaridades 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 áquel 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.**

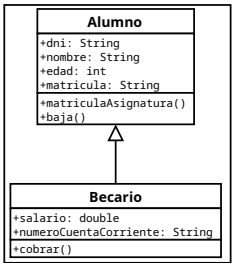
En nuestro ejemplo esta condición implicaría que en cualquier situación en que se pueda participar un `Alumno`, también debería poder participar un `Becario`. Lo complementario no debe porque ser cierto y en la gran mayoría de los casos no lo será.

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 tambien 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.

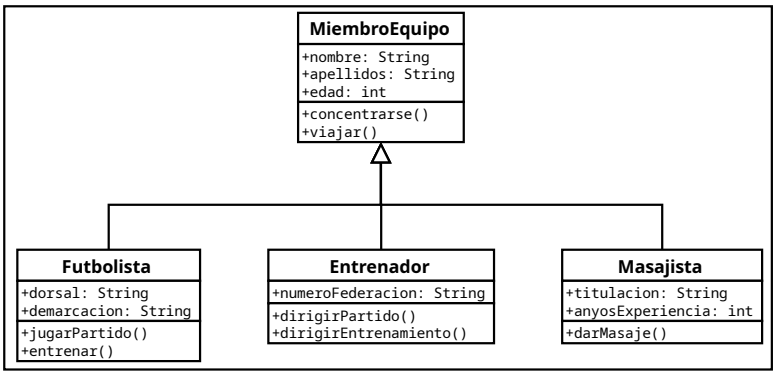
En nuestros ejemplos, los esquemas quedarían como se muestra a continuación.

En el caso del primer esquema:



Aqui 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:

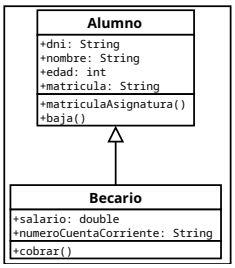


Se puede ver que hemos añadido una nueva clase (`MiembroEquipo`) con los atributos y mensajes comunes a las tres clases previamente existentes (`Futbolista`, `Entrenador` y `Masajista`). En estas clases quedan los atributos y mensajes que son particulares de cada una de estas clases, ya que las comunes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

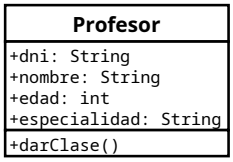
Este concepto se puede extender de forma vertical, es decir, una clase puede heredar de otra clase, que, a su vez, hereda de otra. Este conjunto de relaciones de herencia recuerda a una estructura de árbol, por lo que se llama árbol de herencia. Cuando una subclase hereda de una superclase, hereda no sólo lo que se define explicitamente 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 raiz 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.

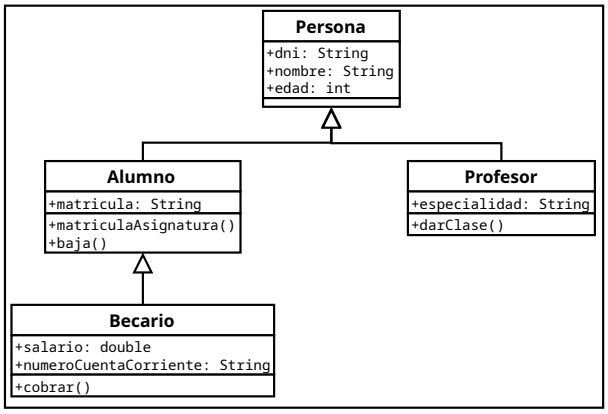
Por ejemplo, retomemos el diagrama con las clases `Alumno` y `Becario`:



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. Este tipo de herencia es más complejo 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.

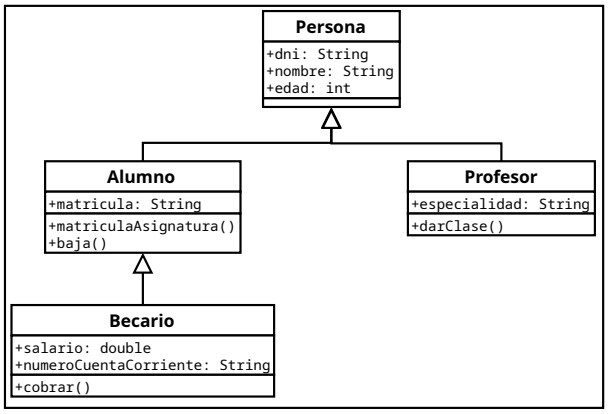
9.1.- Uso de clases heredadas

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.



9.1.1.- 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 aqui que no hayamos visto anteriormente. Lo que si 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 habŕas adivinado ya: La regla que dice que cualquier objeto de una superclase debe poder sustituirse por un objeto de una sublcase.

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 asi 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, asi 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 si 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.

9.1.2.- 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.

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`.

9.1.3.- 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;
```

9.2.- 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 asi 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 excepcion 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 claúsula 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.

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 asi) en los últimos `catch` o, por lo menos, antes que los más específicos (mas subclases). Si no éstos último serán inservibles porque nunca se tomarán.

Por ejemplo, el caso siguiente es correcto:

```
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");
} 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) {
  System.out.println("Fichero no encontrado");
}
```

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");
}
```

10.- Creación de librerías

Crear una librería en Java es relativamente sencillo con las herramientas de que se dispone.

En primer lugar hay que decidir que clases se quieren incluir en la librería. Es importante que se incluyan todas las clases, de las que hemos hecho nosotros, que se necesiten para que el resto funcione. Por ejemplo, si la clase `Alumno` depende de la clase `Curso`, habría que incluir las dos en la librería, aunque la única que nos interese incluir sea `Alumno`, ya que ésta necesita a la otra para funcionar.

También es obligatorio que las librerías estén colocadas en paquetes. Idealmente debería haber un paquete principal de la librería que contenga, o bien directamente las clases de la librería, en caso de un pequeño número de clases, o bien sub-paquetes que organicen las clases.

Una vez decidido que clases queremos incluir, es sencillo crear una librería usando la herramienta JAR.

Podríamos usarla desde la línea de comandos pero Eclipse nos permite crear JAR de una forma sencilla y más rápida que desde la línea de comandos, controlando las dependencias entre clases.

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/)