Programación Orientada A Objetos-Java

- Introducción
- Hello world!
  - Hello world!
  - ¿Qué es Java?
  - Nuestro primer programa
  - Antes de comenzar...
  - ... vamos allá
  - Donde encontrar mas recursos
  - Funciones útiles para el aula
  - Código utilizado en los ejemplos
  - Tarea
- Primeros pasos
  - Primeros pasos
  - Elementos de un programa
  - Separadores
  - Operadores
  - Comentarios
  - Constantes y variables

    - Ámbito
    - Cadenas
    - Arrays
    - Control de flujo
    - Funciones condicionales
    - Bucles
    - Funciones
    - Estructuras de almacenamiento de datos
    - Excepciones
    - Convenios de escritura
    - Algoritmos y estructuras de resolución de problemas sencillos
    - Código utilizado en los ejemplos
    - Tarea

- Introducción a la POO
    - Introducción a la POO
    - Objetos
    - Clases
    - Atributos
    - Constructores
    - Métodos
    - Static
    - Creando objetos
    - Herencia
    - Polimorfismo
    - Sobreescritura de métodos
    - Control de acceso
    - Encapsulamiento
    - Interfaces y clases abstractas
    - Organización del código: Paquetes
    - Código utilizado en los ejemplos
    - Tarea

- Terminando

- Terminando
- Taller con POO
- Identificación de las clases
- Jerarquía de clases
- Codificación
- Interfaz gráfica de usuario
- Calculadora con interfaz gráfica de usuario y ejecutable
- Pintemos en Java
- Bibliografía recomendada
- Código utilizado en los ejemplos
- Tarea
- Voluntario: Muestra tus proyectos
- Créditos

Introducción

```
System.out.println("Hello world!");
```

No podíamos empezar un curso de programación sin la célebre frase *Hello world!*, en la línea superior tenemos la sintaxis que en Java nos permite mostrar por pantalla este mensaje.

A lo largo de este curso de 4 módulos vamos a tratar de familiarizarnos con el lenguaje Java y tener una introducción a la programación orientada a objetos (POO).

Hello world!

Hello world!

En este primer módulo vamos a conocer que es Java y sus características principales. Además vamos a crear nuestro primer programa (lo cual derivará en otras necesidades que veremos entonces) y vamos a conocer donde obtener mas información sobre Java para complementar este curso.

En el siguiente esquema elaborado por el CATEDU podemos ubicar el lenguaje Java como un lenguaje ideal para abarcar el curriculo relativo a programación en alumnado a partir de 15-16 años.

Hello world!

¿Qué es Java?

Java es un lenguaje de programación multiplataforma, de propósito general y orientado a objetos.

En una frase de apenas 14 palabras hemos introducido 4 conceptos:

- **Lenguaje de programación**
- **Multiplataforma**
- **Propósito general**
- **Orientado a objetos**

Vamos a recurrir a la wikipedia para obtener una definición de estos conceptos y tener así una primera aproximación a los mismos

> Un lenguaje de programación es un lenguaje formal diseñado para realizar procesos que pueden ser llevados a cabo por máquinas como las computadoras.
>
> https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n
>
> En informática, multiplataforma es un atributo conferido a programas informáticos o métodos y conceptos de cómputo que son implementados e interoperan en múltiples plataformas informáticas.
>
> https://es.wikipedia.org/wiki/Multiplataforma
>
> Los lenguajes de propósito general, son lenguajes que pueden ser usados para varios propósitos, acceso a bases de datos, comunicación entre computadoras, comunicación entre dispositivos, captura de datos, cálculos matemáticos, diseño de imágenes o páginas.
>
> https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_de_prop%C3%B3sito_general
>
> La programación orientada a objetos (POO, u OOP según sus siglas en inglés) es un paradigma de programación que viene a innovar la forma de obtener resultados. Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

> https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

Con todo lo que hemos visto hasta el momento **tenemos que quedarnos con la idea de que Java es** un lenguaje de programación que va a permitirnos trabajar en distintas plataformas (Windows, Linux, Android,...), con distintas intenciones (hacer un programa que calcule factoriales, acceder a bases de datos, interactuar con el hardware de nuestros dispositivos,...) y además nos va a permitir trabajar con objetos en lugar de hacer uso de la tradicional programación estructurada (que también podremos usar en Java).

Hello world!

Nuestro primer programa

Vista una primera introducción llega el momento de realizar nuestro primer programa.

Una vez se ejecute nuestro primer programa únicamente va a saludarnos por pantalla pero nos va a servir para hablar de cuestiones que hasta el momento hemos obviado y a su vez nos va a servir para configurar nuestro entorno de trabajo.

Hello world!

Antes de comenzar...

Anteriormente hemos comentado que Java es multiplataforma, es decir, un mismo programa puede ser ejecutado en varias arquitecturas o sistemas operativos sin mayor problema. Para conseguir esto es necesario aclarar algunas cuestiones:

- Cuando trabajamos con código java los ficheros que contienen este código tienen la extensión .java Estos ficheros se llaman ficheros fuente y contienen texto plano, es decir, podemos abrirlos con cualquier editor de texto y ver su contenido.
- Cuando un fichero .java es compilado se obtiene un fichero .class del mismo nombre. Si nuestro fichero fuente se llama Saluda.java su versión compilada se llamará Saluda.class y si tratamos de abrirlo con un editor de texto veremos que su contenido nos resulta ilegible

Resumiendo, si yo tengo un fichero fuente ¿qué debo hacer? compilarlo, y para ello necesitarás el kit de desarrollo java (JDK) que incluye el programa *javac* que te permite compilar código. Y entonces, cuando ejecutamos un programa escrito en Java ¿qué estamos ejecutando? estamos ejecutando la versión compilada, es decir, el fichero .class ¿y qué programa interpreta estos ficheros? Lo interpreta un programa llamado *java* que está contenido dentro de la máquina virtual java (JRE) (y que no hay que confundir con las máquinas virtuales que creamos con Virtual Box o similares)

Pregunta Verdadero-Falso

Si quiero poder compilar un fichero .class debo instalar en mi equipo el kit de desarrollo java (JDK)

VerdaderoFalsoLos ficheros que se compilan son los ficheros .java y se compilarían con el programa javac contenido dentro del JDK

Si quiero ejecutar un programa Java del cual me han facilitado su fichero .class en la máquina en la que quiero ejecutarlo deberé tener instalada la máquina virtual java (JRE)

VerdaderoFalso

Si quiero ejecutar un programa Java del cual me han facilitado su fichero .class en la máquina en la que quiero ejecutarlo deberé tener instalada la máquina virtual java (JRE)

VerdaderoFalsoLo que se ejecuta son ficheros .class y se hace con el programa java que está dentro del JRE

Como trabajar en consola con comandos resulta tedioso lo que nosotros vamos a hacer para facilitarnos la vida va a ser trabajar con un entorno de desarrollo integrado (IDE) que nos va a aportar soporte para el lenguaje java evitándonos tener que hacer uso de la consola y facilitándonos el aprendizaje al reseñarnos los errores sintácticos que cometamos.

Para ello he elegido trabajar con el IDE Netbeans que es libre, gratuito y tiene una gran comunidad de gente detrás desarrollándolo y utilizándolo. Entre otras opciones tendríamos el trabajar con Eclipse o IntelliJ IDEA, además de utilizar un editor de texto plano como bloc de notas, notepad++ o Sublime Text y utilizar aparte el terminal.

Netbeans puede ser ejecutado en equipos con sistemas operativos Windows, Linux o Mac OS en sus diferentes arquitecturas de 32 o 64 bits. Para ahorrarnos el descargar por un lado JDK y por otro lado netbeans vamos a descargar e instalar una versión en la que ya viene todo. Para ello nos dirigiremos a http://www.oracle.com/technetwork/articles/javase/jdk-netbeans-jsp-142931.html y descargaremos e instalaremos la versión que se corresponda a nuestro sistema operativo. A continuación he realizado 2 videotutoriales en los que muestro como proceder en Windows 7 y Ubuntu 16.04 LTS.

https://www.youtube.com/embed/DldsDzidscg?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq

https://www.youtube.com/embed/gm2ICqWz4P8?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq

Si queremos prescindir del uso de un entorno de desarrollo integrado podemos hacerlo, aunque yo no lo haría con mi alumnado excepto que trabajase con ellos en FP de la familia de Informática y comunicaciones. En el siguiente enlace tenemos un pequeño manual en el cual nos explican como hacer la compilación y ejecución a través del terminal (previamente habrá que haber instalado el JDK). No obstante a lo largo del curso haré uso de NetBeans en mis explicaciones y vídeos con el fin de facilitar la tarea a docentes y alumnado.

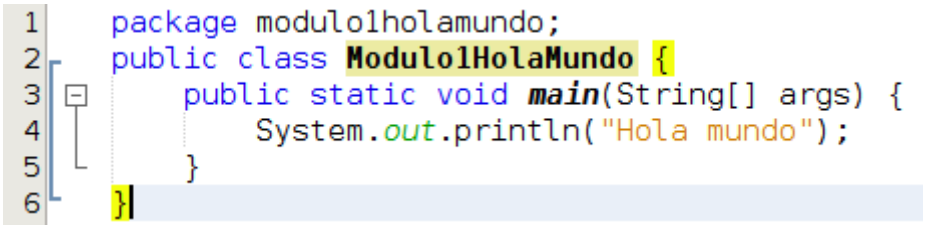
Hello world!

... vamos allá

Como hemos indicado con anterioridad en este curso vamos a utilizar NetBeans como entorno de desarrollo así que ha llegado el momento de familiarizarnos con él y crear nuestro primer programa. En el siguiente videotutorial se muestran aquellas funcionalidades de NetBeans que utilizaremos en el curso y crearemos nuestro primer programa. En este mismo capítulo, con posterioridad al vídeo, desgranaremos este programa.

https://www.youtube.com/embed/plwtjTiiXwo

Voy a extraer el código del programa realizado en el vídeo para comentarlo con mayor detalle. Voy a quitar los comentarios con el fin de facilitar la lectura del código y centrarnos en lo relevante en este momento.



```
1  package modulo1holamundo;
2  public class Modulo1HolaMundo {
3      public static void main(String[] args) {
4          System.out.println("Hola mundo");
5      }
6  }
```

La primera línea hace referencia al **paquete** al que pertenece nuestro fichero. Hablaremos de la organización del código en capítulos posteriores.

En la segunda línea **defino la clase** Modulo1HolaMundo con la palabra reservada class, es importante fijarse que tras la definición de la clase abrimos una llave { que cerramos en la línea 6 } Todo lo contenido entre esas llaves pertenece a la clase. No comento nada de la palabra reservada public ya que la veremos mas adelante pero si es importante reseñar que **el nombre de la clase debe coincidir con el nombre del fichero**. En un mismo fichero puede definirse mas de 1 clase pero solo 1 de ellas será pública.

En la tercera línea creo un **método** llamado main que pertenece a la clase Modulo1HolaMundo. En java para definir un método debemos definir el tipo que devuelve, el nombre y los argumentos que requiere. Posteriormente abrimos una llave { que cerramos en la línea 5 }, todo lo que se encuentra dentro de ambas llaves pertenece al método main En este caso el tipo que devolvemos es void (nada), el nombre de la función es main y tenemos un argumento llamado args de tipo String[]. Los modificadores public y static los dejamos para capítulos posteriores. El **método main es un método especial**, es el que se ejecutará en nuestro programa cuando iniciemos este. En caso de que nuestro proyecto tenga mas de un método main para configurar cual queremos que se ejecute al iniciar el proyecto deberemos dirigirnos a las propiedades del proyecto (properties), seleccionar ejecución (Run) y allí establecer la clase principal (Main Class) que queremos que se ejecute.

En la cuarta línea hago una llamada al sistema y le digo que escriba por pantalla en una línea la frase Hola mundo. Es muy común al principio olvidar el ; del final de la línea.

Hello world!

Donde encontrar mas recursos

Hasta el momento hemos añadido enlaces a distintos recursos de utilidad para aquellas cuestiones que hemos ido tratando.

En este apartado el objetivo es realizar una pequeña recopilación de todos aquellos recursos que puedan resultarnos de utilidad y ampliación a curso. **NO es necesario descargar ni instalar nada de este listado** si has seguido mis instrucciones hasta la fecha.

- Página oficial de Java: https://www.java.com
- Academia Oracle: https://www.java.com/en/about/oracleacademy.jsp en inglés, incluye recursos de formación para estudiantes desde secundaria hasta el mundo de la empresa
- JRE en su versión 8: http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
- JDK en su versión 8: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- API Java 8: https://docs.oracle.com/javase/8/docs/api/
- Tutoriales Java: http://docs.oracle.com/javase/tutorial/
- Aprendiendo el lenguaje: http://docs.oracle.com/javase/tutorial/java/index.html
- Creación de una nterface gráfica: http://docs.oracle.com/javase/tutorial/uiswing/index.html

Funciones útiles para el aula

Cuando hemos creado nuestro primer programa hemos utilizado una función que muestra texto en una línea por pantalla. Voy a recoger en este apartado algunas funciones de la biblioteca de Java (API) que pueden resultarnos de utilidad para los programas que realicemos en el aula:

|Comando|Acción| |:--:|:--:| |System.out.println("Texto");|Escribe un texto y salta a la siguiente línea| |System.out.print("Texto");|Escribe un texto y permanece en la línea| |BufferedReader br = new BufferedReader(new InputStreamReader(System.in));int lado = Integer.parseInt(br.readLine());|Nos permite almacenar en una variable lo que introduzcan en consola| |Math.PI|Equivale a pi| |Math.E|Equivale a e| |Math.pow(base, exponente);|Eleva la base al exponente| |Math.sqrt(número);|Devuelve la raíz cuadrada positiva del número|

En los siguientes capítulos haremos haciendo uso de estas útiles funciones que procuraré ir introduciendo poco a poco pero me ha parecido interesante mostrar un pequeño adelanto.

Hello world!

Código utilizado en los ejemplos

Módulo 1 Proyecto Hola Mundo (zip - 17825 B).

Hello world!

Tarea

Tu tarea una vez acabado este primer módulo consiste en:

- Descargar e instalar Netbeans
- Crear un proyecto llamado Modulo1_Apellidos_Nombre es decir, si te llamas Pablo Ruiz Soria, tu proyecto se deberá llamar Modulo1_Ruiz_Soria_Pablo
- En el proyecto deberá haber un fichero llamado Saluda.java
- Saluda.java debe contener un método principal (main)
- Cuando se ejecute el método principal se deberán escribir por pantalla 2 líneas.
  - En la primera deberá poner tu nombre y apellidos
  - En la segunda línea deberá aparecer "He terminado el primer módulo"

Primeros pasos

Primeros pasos

Hasta el momento hemos realizado una introducción a Java indicando como instalarlo y configurarlo. En este segundo módulo se pretende profundizar en el lenguaje Java y empezar a realizar unos primeros algoritmos que nos permitan introducirnos de lleno en la programación orientada a objetos en el capítulo 3.

Elementos de un programa

En el módulo 1 ya hemos realizado un programa y hemos visto algunas palabras clave y hablado de clases y paquetes pero es ahora cuando vamos a profundizar en las posibilidades del lenguaje Java

Separadores

En Java utilizaremos los siguientes separadores:

- Corchetes []: Se utilizan en operaciones con vectores
- Llaves {}: Sirven para definir bloques de código dentro de las clases y métodos.
- Paréntesis (): Dentro de los paréntesis ubicaremos los parámetros de un método. También nos permiten realizar moldeados de objetos (lo trataremos en el tercer módulo).
- Punto y coma ;: Se utiliza para separar las distintas sentencias de código.
- Punto .: Se utiliza para referirnos a clases y subpaquetes. También nos da acceso a métodos o variables de variables.
- Coma ,: Sirve para separar parámetros dentro de un método. También nos permite separar variables en su declaración.

Con anterioridad ya hemos utilizado todos ellos y conforme avancemos en el curso vamos a ir viendo mas ejemplos de su uso. Vamos a ver un pequeño recordatorio de donde los hemos utilizado:

- Cuando definimos una variable llamada args que era un vector de Strings utilizamos los corchetes: *String**[]**__ args*.
- Cuando definimos la clase Modulo1HolaMundo ya utilizamos las llaves para definir el alcance de la clase: *public class Modulo1HolaMundo **{** (...) **}***.
- Creando creamos el método main hicimos uso para los paréntesis para indicar el comienzo y fin de los parámetros: *public static void main**(**String[] args**)** {*.
- Al finalizar la sentencia *System.out.println("Hola mundo")**;*** hicimos uso del punto y coma para delimitar el fin de la misma.
- En la misma sentencia que en el caso anterior *System**.**out**.**println("Hola mundo");* hicimos uso del punto para acceder a una clase y sus objetos. En este apartado entraremos en mayor profundidad mas adelante.
- El caracter coma aún no lo hemos utilizado, pero podríamos darle uso en el siguiente ejemplo: *void nombreFuncion(int param1**,** int param2);* Aquí la función de la coma no es otra mas que separar los parámetros param1 y para2 del método nombreFuncion.

Pregunta Verdadero-Falso

Para delimitar el contenido de una clase haré uso de los corchetes []

VerdaderoFalsoPara delimitar el contenido de una clase se hace uso de las llaves {} Ocurre lo

mismo con los métodos.

En Java al final de cada línea debo colocar un punto y coma.

VerdaderoFalsoEl punto y coma solo debe colocarse al final de las sentencias. Cuando declaramos una clase o método no hay que hacerlo.

Para crear un vector haré uso de los corchetes []

VerdaderoFalsoPara definir un vector se usan los corchetes []

Operadores

Se hace necesario poder realizar cálculos o tomar decisiones con los datos que manejamos, es por ello que nos surge la necesidad de disponer de operadores que nos permitan hacerlo. A continuación procedo a enumerar los operadores mas habituales:

- = Operando de asignación. Nos permite asignar un valor a una variable.
- + Nos permite sumar números y concatenar cadenas.
- - Nos permite restar números.
- * Nos permite multiplicar números.
- / Nos permite dividir números.
- % Nos permite obtener el resto de una división.
- == Operando de comparación. Devuelve verdadero en caso de que las variables involucradas sean iguales.
- != Operando de comparación. Devuelve verdadero en caso de que las variables involucradas sean distintas.
- < Operando de comparación. Devuelve verdadero en caso de que la variable de la izquierda sea menor a la de la derecha y devuelve falso en cualquier otra situación.
- > Operando de comparación. Devuelve verdadero en caso de que la variable de la derecha sea menor a la de la izquierda y devuelve falso en cualquier otra situación.
- <=Operando de comparación. Devuelve verdadero en caso de que la variable de la izquierda sea menor o igual a la de la derecha y devuelve falso en cualquier otra situación.
- >=Operando de comparación. Devuelve verdadero en caso de que la variable de la derecha sea menor o igual a la de la izquierda y devuelve falso en cualquier otra situación.
- &&: Equivale al operador lógico AND. Devuelve true en caso de que las 2 variables involucradas valgan true, en otro caso devuelven false.
- ||: Equivale al operador lógico OR. Devuelve true en caso de que al menos 1 de las 2 variables involucradas valga true, en otro caso devuelve false.
- variable++ Evalúa el valor de la variable en cuestión y posteriormente le suma 1
- variable-- Evalúa el valor de la variable en cuestión y posteriormente le resta 1
- ++variable Suma 1 a la variable y posteriormente evalúa su valor
- --variable Resta 1 a la variable y posteriormente evalúa su valor

En Java existen mas operadores que los aquí indicados pero creo que con los aquí señalados es mas que suficiente para cubrir los objetivos de este curso. A lo largo del curso se procurará utilizar todos los operadores aquí indicados para así aclarar cualquier posible duda que pudiera existir con ellos.

Comentarios

Los comentarios los introduciremos en nuestro código fuente para facilitar la lectura y comprensión posterior del código. La redacción de estos comentarios también van a facilitar en la posterior elaboración de la documentación del código.

¿Cómo se indica en un programa que algo es un comentario? Existen distintos modos de hacerlo:

- de línea: Todo lo que, en una línea, esté detrás del símbolo // se considerará un comentario y ni el compilador ni el intérprete lo tendrán en cuenta.
- de bloque: Todo lo que esté comprendido entre el símbolo / *y el símbolo* / se considerá un comentario y ni el compilador ni el intérprete lo tendrán en cuenta.
- de documentación: Todo lo que esté comprendido entre el símbolo /* *y el símbolo* / se considerará un comentario de documentación y en la generación de documentación y en la autoayuda del IDE netbeans nos proporcionará información extra.

Vamos a ver un extracto de código que contiene comentarios:

```
230     /**
231      * Método para activar usuarios
232      * @param usuarioEnSession usuario que realiza la acción
233      * @param pass contraseña
234      * @param passRepetido contraseña repetida
235      * @param mail email
236      * @param mailRepetido email repetido
237      * @param lenguajePreferido  identificador del lenguaje elegido por el usuario
238      * @param idAvatar identificador del avatar elegido por el usuario
239      * @param borrable S o N indica respectivamente si el usuario se puede borrar o no
240      * @return devuelve true en caso de actualizar el usuario y falso en caso contrario
241      * @throws TrivinetException
242      */
243     public boolean actualizarUsuario(Usuario usuarioEnSession, String pass,
244             String passRepetido, String mail, String mailRepetido,String lenguajePreferido,
245             Integer idAvatar, String borrable) throws TrivinetException {
246         /* LOG */
247         Util.log("actualizarUsuario("+usuarioEnSession+", pass: ***, passRepetido: ***, " +
248                 ""+mail+", "+mailRepetido+", "+
249                 lenguajePreferido+", " +
250                 "idAvatar: "+idAvatar+", borrable: "+borrable+")");
251         // Comprobaciones
252 
253         // Variables
```

Entre las líneas 230 y 242 tenemos un comentario de documentación. En este tipo de comentarios, además de explicar para que sirve una función se puede detallar información adicional a través de anotaciones como @param @return o @throws que sirven respectivamente para indicar los parámetros de la función, lo que devuelve la función y el tipo de excepción o excepciones que

lanza la función.

En la línea 246 tenemos un ejemplo de comentario de bloque.

En las líneas 251 y 253 tenemos ejemplos de comentarios de línea.

Es muy importante concienciarnos a nosotros/as y a nuestro alumnado de la importancia de documentar el código para que quede reflejada la funcionalidad del código escrito y así facilitar su posterior mantenimiento.

Constantes y variables

Una **[variable](https://es.wikipedia.org/wiki/Variable_(programaci%C3%B3n)** es un lugar en memoria donde almacenaremos un dato que podrá cambiar o no a lo largo de la ejecución de nuestros programas. Todas las variables tienen que ser de un determinado tipo y tener un nombre. Por convenio las variables tienen al menos el primer caracter de su nombre en minúsculas. Una **[constante](https://es.wikipedia.org/wiki/Constante_(inform%C3%A1tica)** es un lugar en memoria en el cual almacenaremos un dato con el fin único de leerlo y sin posibilidad de modificación. Las constante deben tener un determinado tipo y deben tener un nombre. Por convenio el nombre de las constantes se escribe en mayúsculas. En java para definir una constante lo haremos exactamente igual que si definieramos una variable pero las constantes deben ser precedidas por la palabra reservada final.

Anteriormente hemos indicado que tanto las constantes como las variables deben ser de un determinado tipo, a continuación vamos a hablar de ellos.

En java **existen 2 tipos de variables: los tipos básicos** (también llamados tipos primitivos) **y los objetos**. De los objetos nos ocuparemos en el módulo 3 por lo que a continuación paso a enumerar los tipos primitivos mas habituales.

- Números enteros:
  - byte: utiliza 8 bits en memoria. Su rango va de -27 a 27-1
  - short: utiliza 16 bits en memoria. Su rango va de -215 a 215-1
  - int: utiliza 32 bits en memoria. Su rango va de -231 a 231-1
  - long: utiliza 64 bits en memoria. Su rango va de -263 a 263-1
- Números decimales:
  - float: utiliza 32 bits.
  - double: utiliza 64 bits.
- Booleanos:
  - boolean: utiliza 1 bit. Solo puede valer true (verdadero) o false (falso)
- Cadenas:
  - char: utiliza 16 bits.

Ejemplos con tipos primitivos:

```
int operando1 = 4; // Declaramos una variable llamada operando1 de tipo int y le asignamos el valor 4

final double PI = 3.1416; // Declaramos una constante de tipo double. El nombre de la constante es PI y le asignamos el valor 3,1416
```

```
boolean esMayor = false;//Declaramos una varible de tipo boolean, con nombre esMayor y le asignamos el valor falso

final char CARACTER = 'a';// Declaramos una costante de tipo char y nombre CARACTER. Le asignamos el valor 'a'. Fíjate en que son comillas simples.
```

Aunque lo veremos en el siguiente módulo del curso a continuación vamos a crear unas variables que harán referencias a objetos. Del código que aparece a continuación me interesa que nos fijemos en que **declaremos tipos primitivos u objetos la sintaxis es prácticamente la misma**:

```
Integer operando1  = new Integer(4); // Declaramos una variable llamada operando1 de tipo Integer y le asignamos el valor 4

final Double PI = new Double(3.1416); // Declaramos una constante de tipo Double. El nombre de la constante es PI y le asignamos el valor 3,1416

Boolean esMayor = new Boolean(false); // Declaramos una variable de tipo Boolean, con nombre esMayor y le asignamos el valor falso.

final String CADENA = "a";// Declaramos una constante de tipo String y nombre CADENA y le asignamos el valor "a". Fíjate en que son comillas dobles. Además aquí es el único lugar donde no hemos escrito new String("a") pero de esto ya hablaremos mas adelante aunque no es trivial
```

Como podemos intuir en base a estos ejemplos **cada tipo básico tiene asociado un objeto equivalente**. Pero como vemos a continuación (y profundizaremos en el módulo 3) también podemos crear variables del tipo de Objetos que hayamos creado.

```
Coche miCoche = new Coche(); //Declaramos una variable de tipo Coche con el nombre miCoche y llamamos al constructor de la clase. Mas adelante hablaremos de esto

Moto moto1 = null; //Declaramos una variable de tipo Moto con el nombre moto1 y le asignamos el valor null
```

Si eres observador/a te habrás dado cuenta de que los tipos básicos empiezan en minúscula mientras que los objetos comienzan en mayúsculas. Es probable que también te hayas percatado de que el nombre que asigno a las variables siempre comienza en minúsculas y el nombre que asigno a las constantes está completamente en mayúsculas, esto son convenios de escritura del lenguaje Java y tu puedes seguir tu propio estilo de código pero es interesante hacer uso del convenio porque facilita la lectura y comprensión del código yen caso de trabajar con otras personas se es mas productivo. En este módulo, un poco mas adelante, hablaremos en mayor profundidad de esta cuestión.

Ámbito

Las variables y constantes tienen asociado un ámbito. Simplificando, el ámbito indica donde existen y por tanto pueden utilizarse. A continuación voy a tratar de explicarlo con un extracto de código.

```
1  package modulo2ambito;
2  /**
3   * @author Pablo Ruiz Soria
4   */
5  public class Modulo2Ambito {
6      int variableClase = 2;
7      public static void main(String[] args) {
8          int variableMetodo = 2;
9          if(true){
10             int variableCondicion = 4;
11         }
12         for(int i = 0;i < 5; i++){
13             int variableBucle = 5;
14             for(int j = 5; j > 0; j--){
               int variableOtroBucle = 6;
16             }
17         }
18     }
19 }
```

En el código que tenemos en la imagen anterior en la línea 6 declaramos la variable variableClase cuyo ámito es toda la clase, es decir, desde cualquier punto de la clase podríamos acceder a ella. En la línea 8 declaramos la variable variableMetodo, el ámbito de esta variable es el método main, es decir, puede ser accedida desde dentro del main pero no en otros métodos de la clase. En la línea 10 declaramos otra variable llamada variableCondicion cuyo ámbito es el bucle if, no es accesible desde ningún otro lugar. En la línea 12 declaramos la variable de nombre i que es accesible dentro de todo el bucle. En la línea 13 declaramos la variable variableBucle con la que ocurre lo mismo que con i. En la línea 14 declaramos la variable j cuyo ámbito es el bucle anidado y es únicamente accesible desde este bucle pero no del otro. Lo mismo ocurre en la línea 15 con la variable variableOtroBucle.

Pregunta Verdadero-Falso

En el ejemplo de código anterior, podría sustituir la línea 10 por System.out.println(variableClase); ya que la variable variableClase es accesible desde la línea 10

VerdaderoFalsovariableClase es accesible desde toda la clase Modulo2Ambito y la línea 10 está dentro de la clase por lo que es accesible.

En el ejemplo de código anterior, podría añadir al final de la línea 6 System.out.println(variableBucle); ya que la variable variableBucle es accesible desde la línea 6

VerdaderoFalsovariableBucle solo está accesible dentro del bucle que está entre las líneas 12 y 17 por lo que no es accesible en la línea 6 y obtendríamos un error de compilación.

Cadenas

Hasta el momento hemos trabajado con cadenas (**String**) en alguna ocasión. Hemos mostrado por pantalla una cadena:

```
System.out.println("Hello world!");
```

Hemos definido una constante

```
final String CADENA = "a";// Declaramos una constante de tipo String y nombre CADENA y le asignamos el valor "a". Fíjate en que son comillas dobles.
```

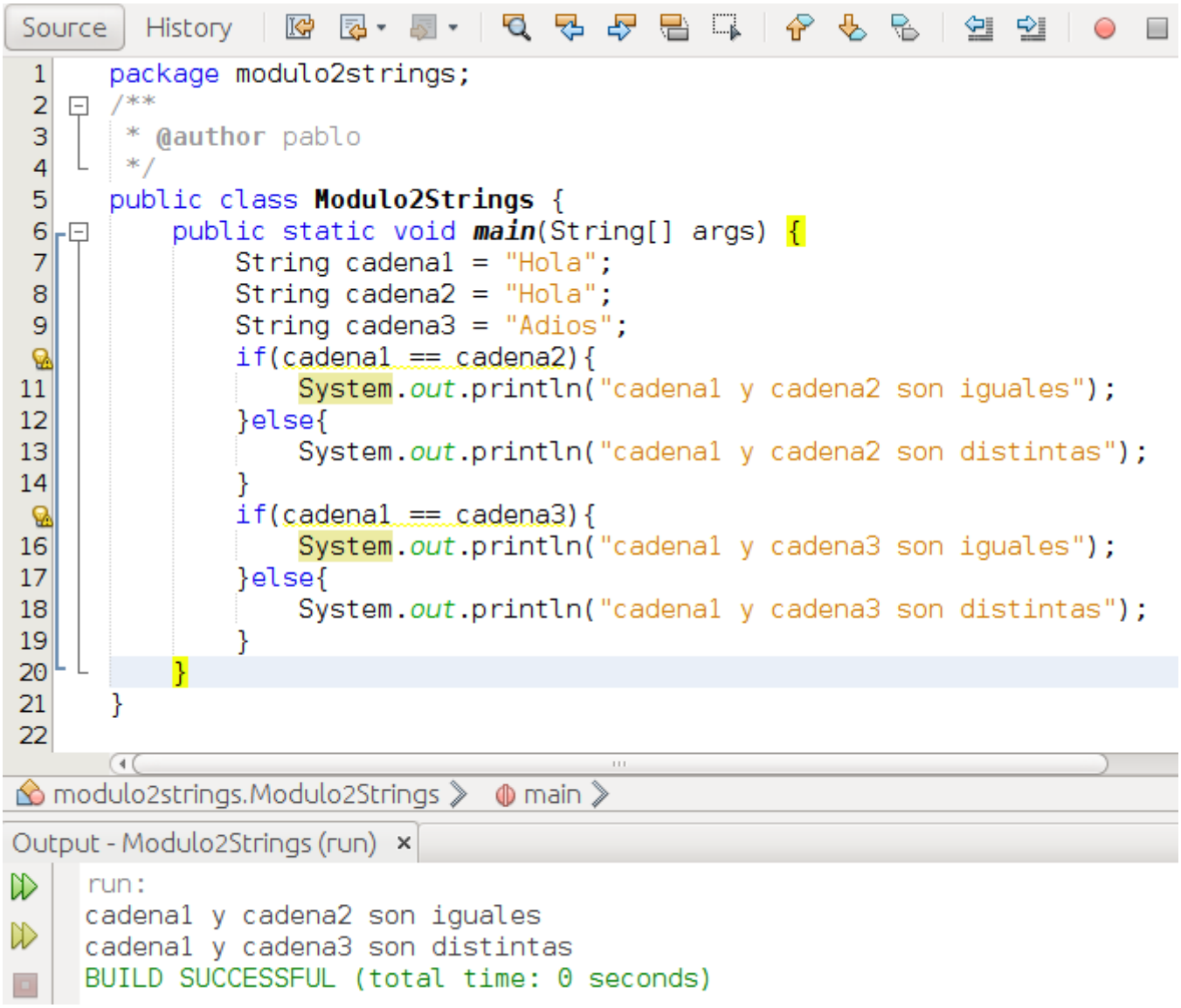
y del mismo modo podríamos haber creado una variable:

```
String texto = "Hola mundo";// Declaramos una variablede tipo String y nombre textoy le asignamos el valor "Hola mundo". Fíjate en que son comillas dobles.
```

String no es un tipo primitivo, String es una clase por lo que cada vez que hacemos uso de ella estamos creando un objeto.

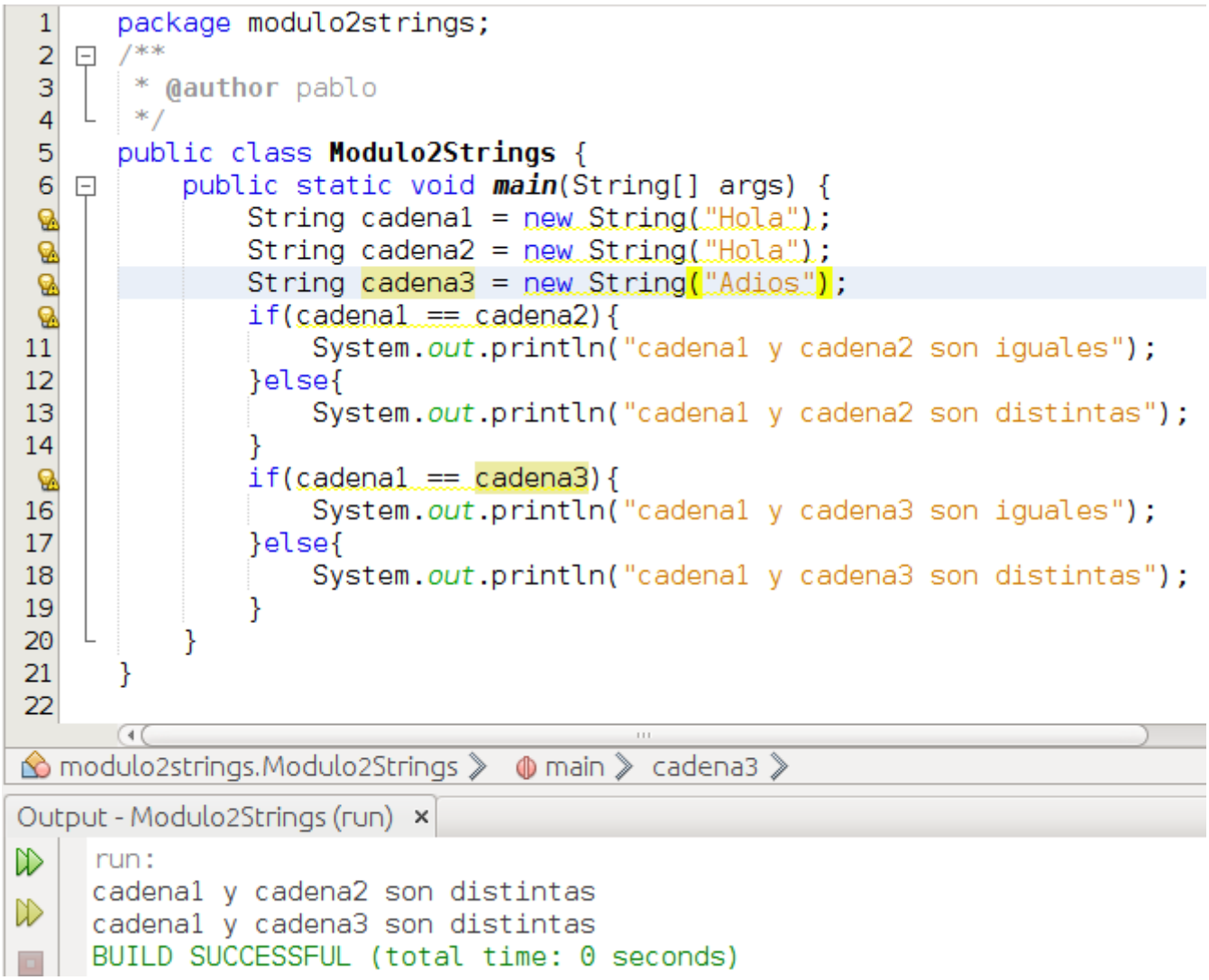
Como ya vimos en este módulo en el apartado de constantes y variables y como veremos en el siguiente módulo de este curso que cada vez que creamos un objeto lo hacemos con la sintaxis *Clase nombreVariable = new Clase();*. Sin embargo según acabo de indicar si utilizamos String estamos creando un objeto ¿entonces porque aquí no usamos new y hacemos la asignación directamente? Vamos a ver un par de ejemplos:

Ejemplo 1, sin new:



```
package modulo2strings;
/**
 * @author pablo
 */
public class Modulo2Strings {
    public static void main(String[] args) {
        String cadena1 = "Hola";
        String cadena2 = "Hola";
        String cadena3 = "Adios";
        if(cadena1 == cadena2){
            System.out.println("cadena1 y cadena2 son iguales");
        }else{
            System.out.println("cadena1 y cadena2 son distintas");
        }
        if(cadena1 == cadena3){
            System.out.println("cadena1 y cadena3 son iguales");
        }else{
            System.out.println("cadena1 y cadena3 son distintas");
        }
    }
}
```


Ejemplo2, con new:



```
1  package modulo2strings;
2  /**
3   * @author pablo
4   */
5  public class Modulo2Strings {
6      public static void main(String[] args) {
           String cadena1 = new String("Hola");
           String cadena2 = new String("Hola");
           String cadena3 = new String("Adios");
           if(cadena1 == cadena2){
               System.out.println("cadena1 y cadena2 son iguales");
11         }else{
12             System.out.println("cadena1 y cadena2 son distintas");
13         }
14         if(cadena1 == cadena3){
15             System.out.println("cadena1 y cadena3 son iguales");
16         }else{
17             System.out.println("cadena1 y cadena3 son distintas");
18         }
19     }
20 }
21
22
```

modulo2strings.Modulo2Strings > main > cadena3 >

Output - Modulo2Strings (run)

```
run:
cadena1 y cadena2 son distintas
cadena1 y cadena3 son distintas
BUILD SUCCESSFUL (total time: 0 seconds)
```

En el primer ejemplo, sin new, lo que ocurre es que cuando creamos la variable cadena1 se crea un espacio en memoria con el valor "Hola" y cuando creamos una nueva variable llamada cadena2 y decimos que valga "Hola" en vez de crearse un nuevo espacio en memoria se apunta al mismo, por eso, cuando se hace la comparación con == se comparan las zonas de memoria y al ser iguales nos dice que cadena1 es igual a cadena2. Sin embargo, en el 2º caso, al crear las variables cadena1 y cadena2 utilizando new estamos forzando a que cada variable (objeto) ocupe una zona de memoria distinta aunque tengan el mismo valor. Puede resultar confuso pero creo conveniente reseñar este caso especial ya que al tratarse de un objeto es mas que probable que nuestro alumnado nos pregunte por esta peculiaridad de este objeto. Esto no ocurre con ningún otro objeto.

En un entorno profesional no se usaría la clase String sino que se usarían las clases StringBuffer o StringBuilder (en función de las necesidades específicas). En el curso utilizaremos la clase String pero por curiosidad vamos a ver un ejemplo de código con estas clases:

```java
package modulo2strings;
/**
 * @author pablo
 */
public class Modulo2Strings {
    public static void main(String[] args) {
        StringBuffer cadena1 = new StringBuffer();
        cadena1.append("¡Hola ");
        cadena1.append("mundo!");
        StringBuilder cadena2 = new StringBuilder();
        cadena2.append("Hello ");
        cadena2.append("world!");
        System.out.println(cadena1);
        System.out.println(cadena2);
    }
}

```

modulo2strings.Modulo2Strings > main >

Output - Modulo2Strings (run) ×

```
run:
¡Hola mundo!
Hello world!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Arrays

En Java existen un tipo de variables llamadas * ***que nos permiten crear*** ***vectores** de un mismo elemento. Es importante indicar que un Array es un objeto.

Existen distintos modos de declarar un Array, vamos a verlos:

```
int[] array1 = new int[4]; // Creamos una variable llamada array1 que tendrá 4 posiciones
int array2[] = new int[4]; // Creamos una variable llamada array2 que tendrá 4 posiciones
int[] array3 = {1, 2, 3}; // Creamos una variable llamada array3 que en su posición 0 tendrá un 1, en su posición 1 tendrá un 2 y en la posición 2 tendrá un 3
```

Cuando creamos un Array como es el caso de array1 y array2 y lo inicializamos pero no definimos el valor de sus posiciones estas toman el valor por defecto de la variable, en nuestro caso 0.

La primera posición de un Array es la posición 0. En caso de intentar acceder a una posición inexistente obtendremos una excepción (se tratan mas adelante)

De un modo análogo al anterior podemos crear de Arrayx (matrices). A continuación podemos ver un ejemplo:

```
int[][] matriz = new int[10][10];// Creamos una variable llamada matriz que es una matriz de enteros de 10x10
```

Si queremos acceder al valor de un array debemos hacerlo con la sintaxis nombreVariable[posicion]. A continuación unos ejemplos:

```
boolean[] array4 = {false, true, false};
System.out.println(array4[0]);
System.out.println(array4[1]);
System.out.println(array4[2]);
```

Y lo que veríamos por pantalla:

```
false
true
false
```

Cuando lleguemos al apartado de control de flujo realizaremos mas ejemplos con .

Control de flujo

Además de poder operar con los datos tenemos la necesidad de que ocurran cosas distintas en función de los mismos. Para cubrir estas necesidades dispondremos de funciones condiciones y bucles.

Como funciones condicionales trabajaremos con:

- if/else if/else
- switch

Como bucles trabajaremos con:

- for
- while
- do while

En los nodo inferiores voy a tratar de explicarlos y veremos algunos ejemplos con código.

Funciones condicionales

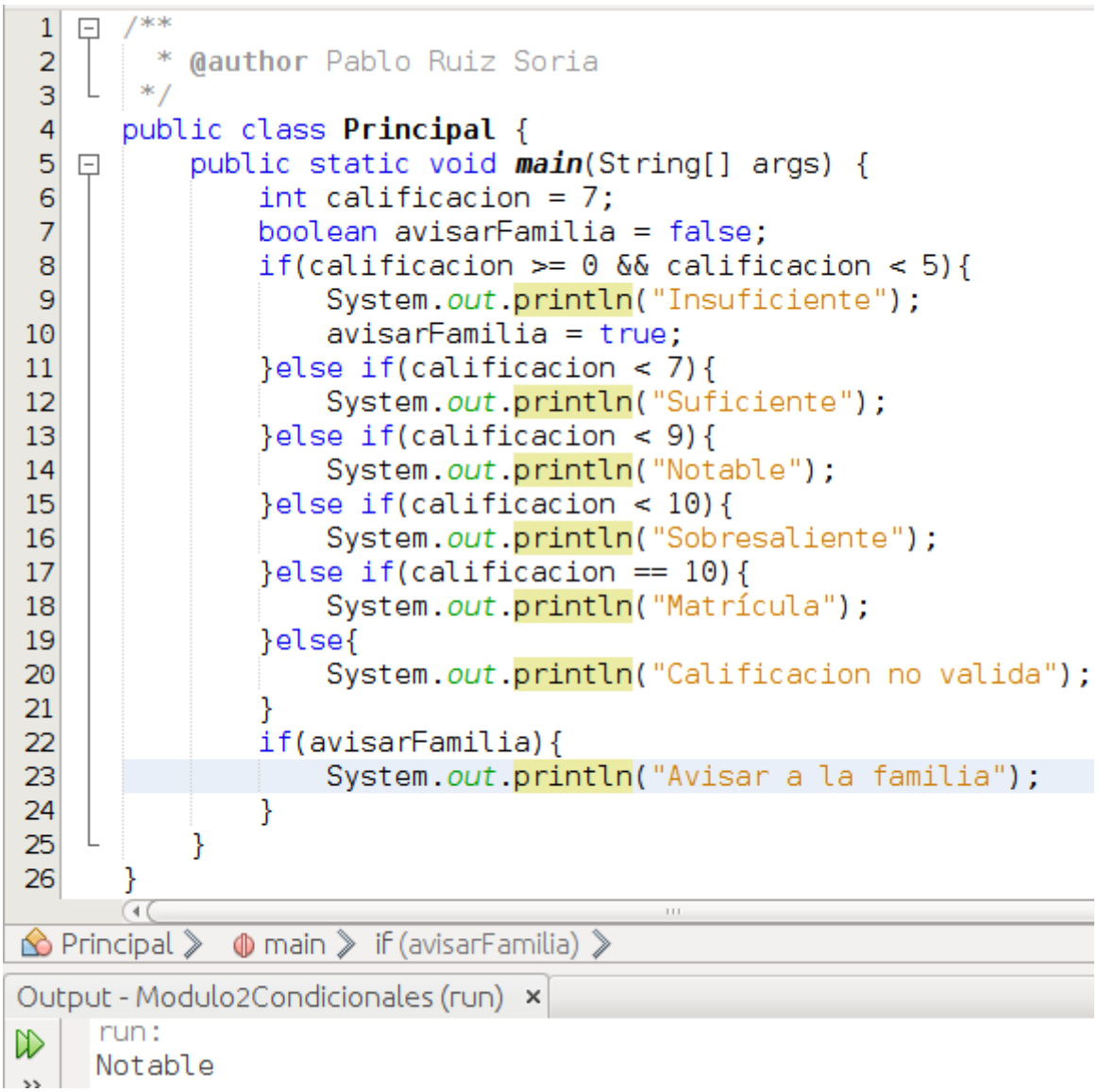
Existen 2 funciones condicionales: **if/else if/else** y **switch**.

Vamos a comenzar por la estructura **if/else if/else**. En esta estructura la sintaxis es la siguiente:

```
if(condicion){
    sentencias;
}else if(condicion){
    sentencias;
}else{
    sentencias;
}
```

De la estructura anterior es obligatorio que exista siempre el if. else if puede haber 0 o varios. else puede haber 1 o 0. Y el funcionamiento es el siguiente. Cuando el compilador llega a la condición del if y la evalúa, si la condición es verdadera ejecutará las sentencias ubicadas dentro del if y habrá terminado con la estructura if/else if/else, ya no ejecutará sentencias de otros bloques aunque estas sean verdad. Si la condición del if es falsa se evalúa la condición del else if y si es verdadera se ejecutan las sentencias ubicadas dentro, en caso de ser falsa se evalúa el siguiente else if si lo hubiera. Si la condición de algún else if fuese verdadera se ejecutarían las sentencias de su interior y ya no se evaluarían ni ejecutarían mas condiciones de toda la estructura if/else if/else. En caso de que ninguna condición del if o los else if fuese verdadera y hubiese un bloque else se ejecutarían las sentencias contenidas dentro del bloque else.

Vamos a ver a continuación un ejemplo de 2 estructuras if/else if/else ya que es mas sencillo comprender esta estructura leyendo el código que tratando de explicarlo.



```java
/**
 * @author Pablo Ruiz Soria
 */
public class Principal {
    public static void main(String[] args) {
        int calificacion = 7;
        boolean avisarFamilia = false;
        if(calificacion >= 0 && calificacion < 5){
            System.out.println("Insuficiente");
            avisarFamilia = true;
        }else if(calificacion < 7){
            System.out.println("Suficiente");
        }else if(calificacion < 9){
            System.out.println("Notable");
        }else if(calificacion < 10){
            System.out.println("Sobresaliente");
        }else if(calificacion == 10){
            System.out.println("Matrícula");
        }else{
            System.out.println("Calificacion no valida");
        }
        if(avisarFamilia){
            System.out.println("Avisar a la familia");
        }
    }
}
```


Cuando el programa llega al if de la línea 8 se comprueba si la variable calificacion es mayor o igual a 0 (lo es) y que la variable calificacion sea menor a 5 (no lo es). Como el resultado de true && false es false no entramos dentro de sus llaves y se pasa a evaluar la condición de la línea 11. En la línea 11 se evalúa si la variable calificacion es menor a 7 (no lo es), al ser falsa se pasa a evaluar la condición de la línea 13. En la línea 13 se evalúa si la variable calificacion es menor a 9 (lo es), al ser verdad se ejecutan todas las sentencias ubicadas dentro de sus llaves, en ese caso solo hay 1 sentencia y se escribe Notable en consola. Como la condición de la línea 13 es verdad ya no se comprueban las condiciones de las líneas 15, 17 o 19 y se salta directamente a la línea 22.

En la línea 22 tenemos otra condición así que se evalúa si la variable avisarFamilia es true (no lo es), al ser falso no se ejecuta el código de su interior y al no haber ningún else if ni else se da por terminado este if.

Existe también la posibilidad de ejecutar esta estructura con la sintaxis:

```
condicion?valorSiCondicionVerdadera:valorSiCondicionFalsa;
```

Vamos a ver un ejemplo que sustituiría al código comprendido entre las líneas 22 a 24 de la imagen anterior

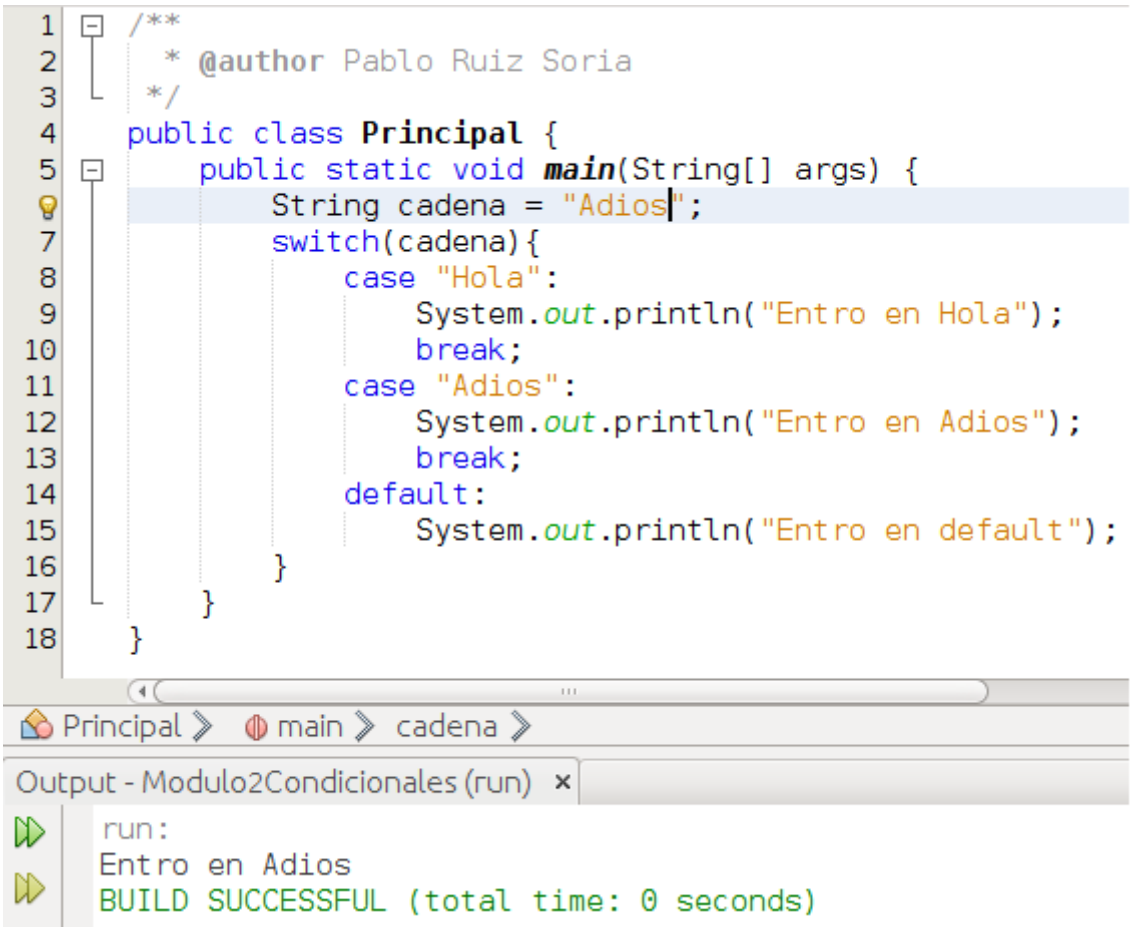
```
22          System.out.println(avisarFamilia?"Avisar a la familia":"");
```

Se va a escribir algo por pantalla ¿el qué? dependerá del resultado que produzca la variable avisarFamilia, si la variable avisarFamilia es verdadero se devolverá "Avisar a la familia" y será eso lo que se escriba. Si la variable avisarFamilia es falso se devolverá lo que está tras los 2 puntos, es decir, "" y se escribirá una línea vacía.

Una vez vista la estructura if/else if/else vamos a continuar con la estructura **switch**. Su sintaxis es la siguiente:

```
switch(expresión){
    case valor1:
        Sentencias;
        break;
    case valor2:
        Sentencias;
        break;
    default:
        Sentencias;
}
```

De la sintaxis anterior es obligatorio utilizar la primera línea y la última y debe haber al menos 1 case con sus sentencias y un break tras las sentencias. El elemento default es opcional. Como quizás resulte algo complejo de explicar teóricamente vamos a ver un ejemplo:



En la línea 7 indicamos a nuestro switch que la variable de referencia es cadena. En la línea 8 comprobamos si "Hola" es igual a cadena y como no lo es pasamos al siguiente case, al de la línea 11. En la línea 11 se evalúa si la variable cadena es igual a "Adios" y como lo es entramos a ejecutar todas las sentencias ubicadas dentro de nuestro case, en este caso la de la línea 12. Luego llegamos a la línea 13, donde nos encontramos un break, este break nos lanza fuera del switch, a la línea 16. Si olvidásemos añadir el break de la línea 13 se ejecutaría la sentencia de la línea 12 y luego la sentencia de la línea 15 por ello hay que finalizar cada case con break.

Si hablamos de la **eficiencia interna** de cada una de estas estructuras condicionales la documentación de Java indica que es mas óptimo hacer uso de switch si bien es cierto que en los desarrollos que realicemos en el aula con nuestro alumnado no vamos a notar la diferencia de rendimiento.

Bucles

Ha llegado el momento de hablar de los bucles y de sus distintas formas en Java.

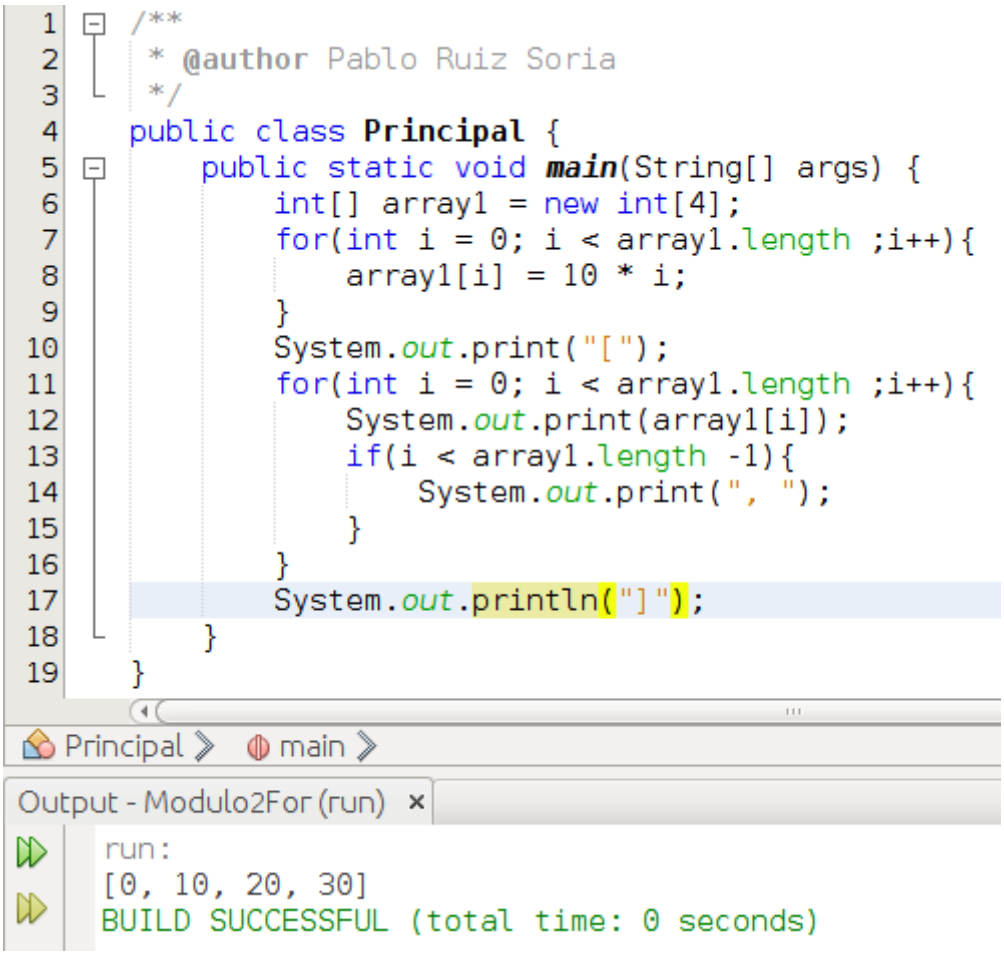
Lo primero es definir que es un bucle, en la wikipedia podemos encontrar la siguiente definición de bucle:

> Un bucle o ciclo, en programación , es una sentencia que se realiza repetidas veces a un trozo aislado de código, hasta que la condición asignada a dicho bucle deje de cumplirse.
>
> https://es.wikipedia.org/wiki/Bucle_(programaci%C3%B3n)

Ahora que sabemos lo que es vamos a ver los distintos bucles existentes en Java. Vamos a comenzar con el bucle **for**. Su sintaxis es la siguiente:

```
for(iniciacion; condicionFin; accionSobreIniciacion){
    sentencias;
}
```

En iniciacion deberemos establecer el valor inicial que queremos dar a nuestra variable de control. En condicionFin deberá haber alguna condición lógica y en accionSobreIniciacion deberemos establecer que queremos que pase tras cada iteracción. En sentencias estableceremos las sentencias de código que necesitemos. No se considera una buena práctica modificar la variable de iniciación en las sentencias. Una vez mas, vamos a reforzar la explicación con un ejemplo que incluya código con el fin de facilitar la compresión.



```java
/**
 * @author Pablo Ruiz Soria
 */
public class Principal {
    public static void main(String[] args) {
        int[] array1 = new int[4];
        for(int i = 0; i < array1.length ;i++){
            array1[i] = 10 * i;
        }
        System.out.print("[");
        for(int i = 0; i < array1.length ;i++){
            System.out.print(array1[i]);
            if(i < array1.length -1){
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }
}
```

Principal > main >

Output - Modulo2For (run)

```
run:
[0, 10, 20, 30]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Vamos a analizar esta pieza de código. En la línea 6 creamos un array de 4 posiciones (tal como vimos en el apartado Arrays). En la línea 7 tenemos un bucle for y en el nos encontramos int i = 0, esto hace que solo la primera vez que se accede al bucle se cree una variable llamada i que valga 0. Posteriormente se evalúa la condición array1.length (nos devuelve la longitud del array) y como es verdad (0 es menor que 4) se procede a ejecutar las sentencias del interior del for. En este caso se trata de una única sentencia que lo que hace es establecer un valor en la posición del array. Una vez se han ejecutado todas las sentencias de dentro de bucle for se ejecuta i++ y se reevalúa la condición, si esta es cierta se vuelven a ejecutar las sentencias del bucle y así hasta el momento en que la condición sea falsa, momento en el cual se sale del bucle. En la línea 10 escribimos por pantalla un corchete y posteriormente, en el bucle, recorremos las posiciones del array mostrando su valor y además en caso de que no se trate de la última posición del array añadimos una coma. Cuando termina el bucle se añade el cirre del corchete antes escrito. Voy a escribir lo que ocurre en el primer bucle for de un modo menos prosaico a continuación:

1º Se crea una variable llamada i y se le asigna el valor 0

2º Se comprueba que la variable i (que vale 0) sea menor a array1.length (que vale 4). Es verdad.

3º Ejecuto array1[i] = 10 * i; es decir, en la posición 0 del array1 pongo el resultado de multiplicar 10 * 0.

4º Hago i++. i pasa a valer 1.

5º Se comprueba que la variable i (que vale 1) sea menor a array1.length (que vale 4). Es verdad.

6º Ejecuto array1[i] = 10 * i; es decir, en la posición 1 del array1 pongo el resultado de multiplicar 10 * 1.

7º Hago i++. i pasa a valer 2.

8º Se comprueba que la variable i (que vale 2) sea menor a array1.length (que vale 4). Es verdad.

9º Ejecuto array1[i] = 10 * i; es decir, en la posición 2 del array1 pongo el resultado de multiplicar 10 * 2.

10º Hago i++. i pasa a valer 3.

11º Se comprueba que la variable i (que vale 3) sea menor a array1.length (que vale 4). Es verdad.

12º Ejecuto array1[i] = 10 * i; es decir, en la posición 3 del array1 pongo el resultado de multiplicar 10 * 3.

13º Hago i++. i pasa a valer 4.

14º Se comprueba que la variable i (que vale 4) sea menor a array1.length (que vale 4). Es falso, se sale del bucle.

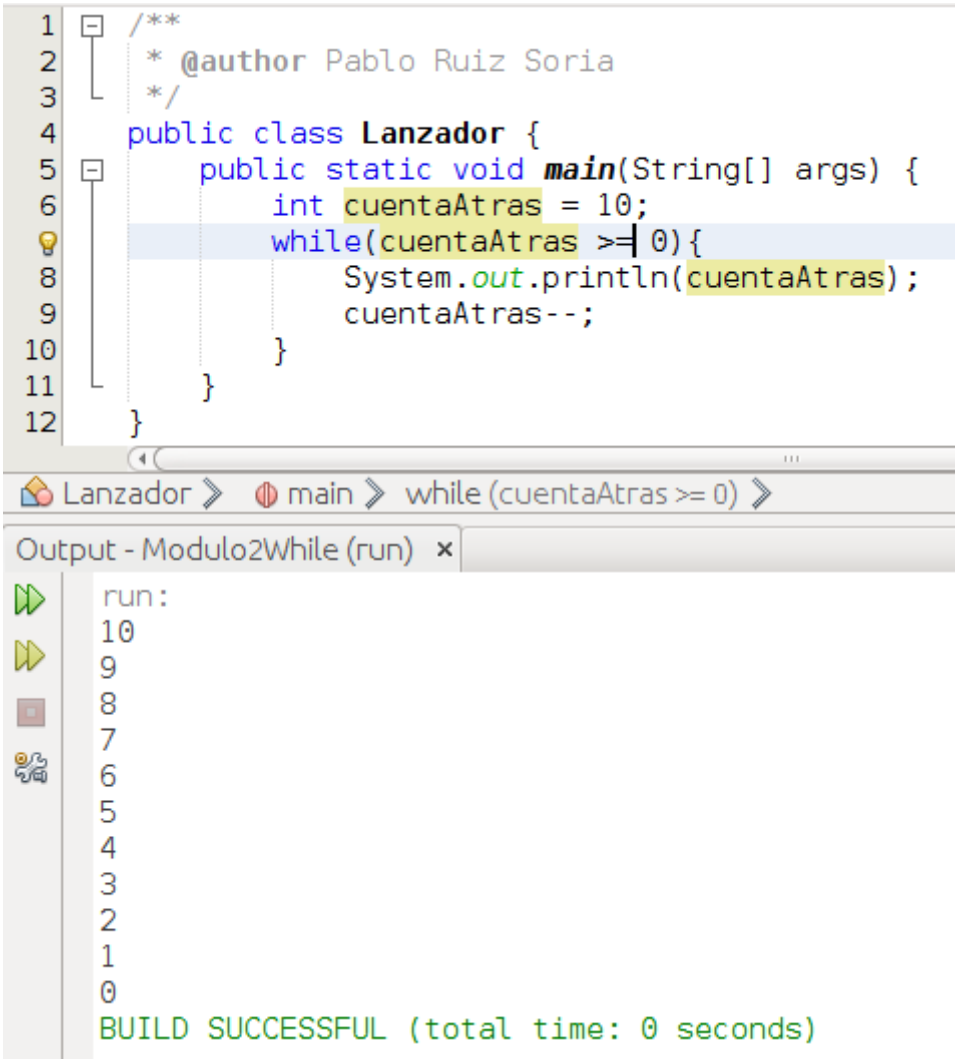
En Java existe otra manera de trabajar con bucles for, vamos a ver la sintaxis a continuación pero veremos ejemplos en el apartado de estructuras de almacenamiento de datos.

for(tipoVariable nombreVariable : listaQueContieneVariablesDeTipoVariable){ sentencias; }

Una vez hemos visto las posibilidades que nos ofrece el bucle for vamos a ver los **bucles while**. Los bucles while tienen la siguiente sintaxis:

```
while(condicion){
    sentencias;
}
```

En un bucle while las sentencias de su interior se van a estar ejecutando siempre mientras la condición del while sea verdadera. Por ello, salvo que queramos hacer un bucle infinito, en las sentencias actuaremos sobre la variable o variables que participen en la condición. Vamos a ver un ejemplo:

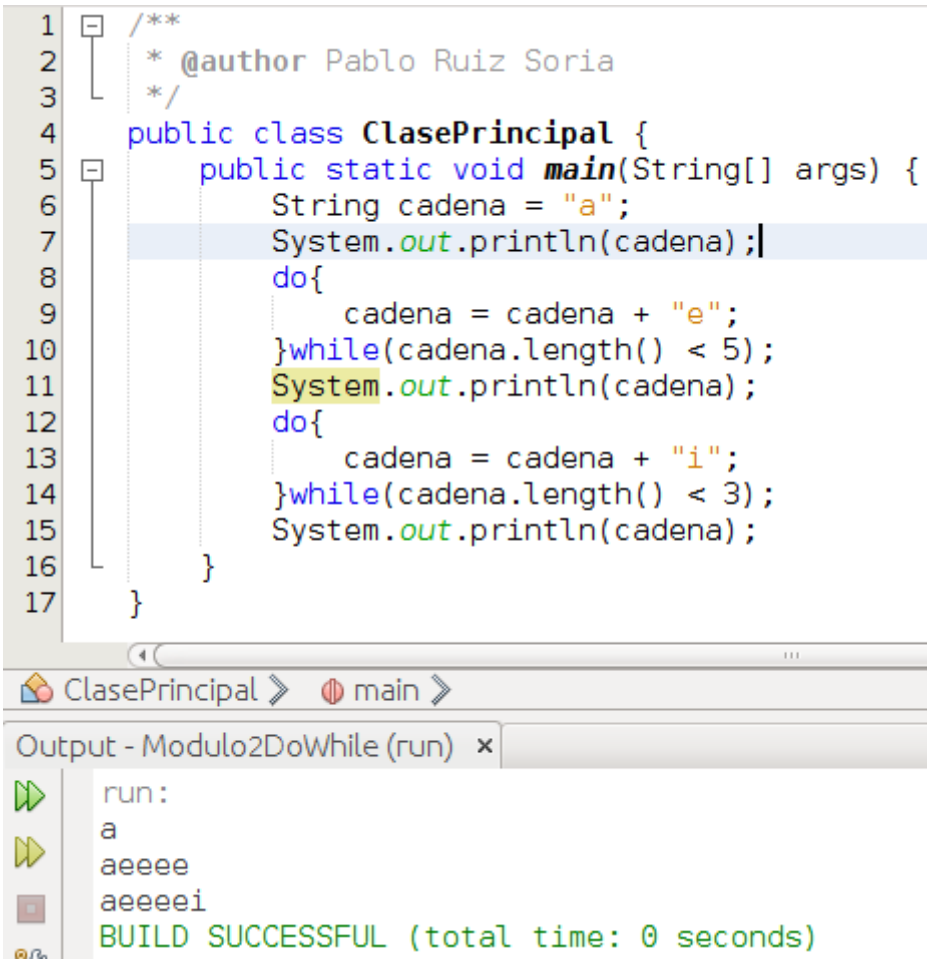


Lo que ocurre en el código de la imagen es lo siguiente. En la línea 6 creamos una variable de tipo entero con valor 10. En la línea 7 se evalúa que la variable antes creada sea mayor o igual a 0. Si es verdad, se ejecutan las sentencias de las líneas 8 y 9. Si es falso, se sale del bucle while.

En último lugar vamos a hablar de los bucles **do-while**. Estos bucles tienen la siguiente sintaxis:

```
do{
   sentencias;
}while(condicion);
```

Los bucles do-while son prácticamente iguales a los bucles while con la salvedad de que en un bucle do-while las sentencias contenidas dentro del mismo se van a ejecutar siempre 1 vez y luego se seguirán ejecutando mientras la condición sea verdadera. Al igual que en el caso anterior, salvo que queramos hacer un bucle infinito, en las sentencias deberemos actuar sobre la variable o variables que participen en la condición.



Vamos a analizar que sucede en este código. En la línea 6 creamos una variable de tipo String con el valor a. En la línea 7 mostramos en una línea el valor de nuestra variable cadena. Entre las líneas 8 y 10 nos encontramos un bucle do-while, procedemos ejecutando todo lo que está dentro del bucle por lo que a nuestra variable cadena le añadimos una e al final, ahora evaluamos la condición. Si la condición es cierta volvemos a ejecutar todas las sentencias contenidas dentro del do-while. Si la condición es falsa hemos terminado con el do-while. En la línea 11 mostramos en una línea el valor de nuestra variable cadena. Entre las líneas 12 y 14 nos encontramos otro bucle do-while, así que antes de nada ejecutamos las sentencias de su interior, en este caso a nuestra variable cadena le añadimos una i al final y una vez hemos ejecutado las sentencias procedemos a evaluar la condición.Si la condición es cierta volvemos a ejecutar todas las sentencias contenidas dentro del do-while. Si la condición es falsa hemos terminado con el do-while. En la línea 15 mostramos en una línea el valor de nuestra variable cadena.

Presta atención a este detalle: Para conocer el tamaño de un array utilizamos nombreVariableDeTipoArray.length mientras que para conocer la longitud de una cadena utilizamos nombreVariableTipoString.length(). La diferencia radica en el uso del paréntesis. Es algo que en nuestros primeros pasos con Java puede ocasionarnos algún pequeño trastorno.

Funciones

En este apartado vamos a hablar de las **funciones**, también llamadas **procedimientos**, y vamos a nombrar los **métodos** pero estos últimos los veremos en mayor profundidad en el módulo 3.

Una función es un trozo de código definido por un nombre a la cual se le pueden pasar parámetros o no y que puede o no devolver un valor.

Como primera aproximación a los métodos vamos a indicar que son algo muy parecido a una función pero ligado a un objeto. En el siguiente módulo profundizaremos en los métodos.

Vamos a ver unos trozos de códigos para comentarlos mas adelante:

Código con ejemplo de cálculo de medias

Ahora vamos a ver como hacer esto con una función:

Código con ejemplo de cálculo de medias con funciones

Y vamos a ver que hacemos y porqué. Lo primero es indicar que ambos códigos hacen lo mismo como podemos ver en la salida del programa sin embargo en la 2ª imagen entre las líneas 6 y 8 creamos una función que devuelve un float, tiene por nombre hazMedia y necesita 2 parámetros de tipo float. Esta función realiza la media de ambos datos y la retorna. En el segundo programa, en las líneas 13 y 17, vemos que para llamar a esta función es suficiente con indicar a que variable vamos a asignarle el valor devuelto y facilitar las variables con las que queremos que se haga el cálculo. Este ejemplo es trivial, pero si pensásemos en funciones mas complejas veríamos que al sacar factor común del código repetido este queda mucho mas simple y legible, además, si utilizamos una función estamos siguiendo el principio DRY (Don't Repeat Yourself, no te respitas) que nos va a permitir el que en caso de tener que realizar un cambio en el algoritmo únicamente haya que hacerlo en la función y no en todos los sitios donde aparezca el código no llevado a la función.

Vamos a ver ahora el código de una función que no devuelve nada:

Codigo con ejemplo de uso de funciones y procedimientos

En Java podemos hacer uso de la recursividad en las funciones, a continuación tenemos un ejemplo:

Ejemplo de código con función recursiva

Estructuras de almacenamiento de datos

Antes de comenzar vamos a conocer la definición que de **estructura de datos** ofrece la wikipedia:

> En programación, una estructura de datos es una forma particular de organizar datos en una computadora para que pueda ser utilizado de manera eficiente.
>
> https://es.wikipedia.org/wiki/Estructura_de_datos

En capítulos anteriores ya hemos trabajado con una estructura de datos, los **Arrays** (los cuales nos permitían almacenar datos en **vectores**).

Trabajar con Arrays puede ser suficiente para nuestras prácticas de aula pero conviene conocer las interfaces List, Map y Set. En el siguiente módulo del curso veremos que es una interface pero ahora nos interesan conocer algunas implementaciones de las interfaces antes mencionadas. ArrayList, HashMap y HashSet son, respectivamente, algunas de las implementaciones de estas interfaces.

Vamos a ver para que usar cada una de estas estructuras de datos:

- **Arrays (vectores)**: Es la forma mas eficiente de almacenar objetos pero una vez defines el tamaño del vector no puedes ampliarlo o reducirlo. Además no puedes guardar variables de distinto tipo
- **List**: Almacena las variables en el orden en que se insertan. Nos permite tener valores duplicados en la lista. Nos permite tener variables de distinto tipo (al declararla no pondremos como en nuestro ejemplo)
- **Map**: No almacena el orden en que se insertan los datos (algunas de sus implementaciones si lo hacen). Para almacenar los datos se hacen usando el par clave-valor. No permite valores de clave repetidos pero si valores de valor repetidos. Nos permite tener variables de distinto tipo.
- **Set**: No almacena el orden en que se insertan los datos (algunas de sus implementaciones si lo hacen). No permite valores duplicados. Nos permite tener variables de distinto tipo. Es lo que debemos elegir si no queremos tener elementos repetidos en nuestra estructura de datos.

En este capítulos vamos a centrarnos en la clase **ArrayList**, vamos a ver un ejemplo:

```
1  import java.util.ArrayList;
2  /**
3   * @author Pablo Ruiz Soria
4   */
5  public class ClasePrincipal {
6      public static void main(String[] args) {
7          String[] vectorDeStrings = new String[2];
8          for(int i = 0; i < vectorDeStrings.length ;i++){
9              vectorDeStrings[i] = "Hola mundo";
10         }
12         for(int i = 0; i < vectorDeStrings.length ;i++){
13             System.out.println(vectorDeStrings[i]);
14         }
15         ArrayList<String> listadeStrings = new ArrayList();
16         for(int i = 0; i < 2 ;i++){
17             listadeStrings.add("Hello world");
18         }
19         for(int i = 0; i < listadeStrings.size() ;i++){
20             System.out.println(listadeStrings.get(i));
21         }
22     }
```

```
Output - Modulo2EstructurasDeAlmacenamiento (run)
run:
Hola mundo
Hola mundo
Hello world
Hello world
```

Lo primero que vemos en la línea 1 es que para trabajar con la clase ArrayList hay que importarla. Entre las líneas 7 y 13 tenemos un ejemplo de como trabajar con Arrays (ya lo vimos con anterioridad). En la línea 14 nos encontramos con la creación de una variable llamada listadeStrings que es de tipo ArrayList y además le añadimos lo cual significa que en este ArrayList solo vamos a poder almacenar variables de tipo String. La primera diferencia con respecto a los Arrays es que aquí no definimos el tamaño del ArrayList. Esto es porque los ArrayList, a diferencia de los Arrays, son dinámicos. Podemos variar su tamaño en tiempo de ejecución según nuestras necesidades. En la línea 16 vemos como añadir un elemento a nuestro ArrayList. En la línea 18 vemos que para obtener el tamaño de un ArrayList utilizamos el método size. Y en la línea 19 vemos que para obtener una determinada posición de un ArrayList utilizamos el método get. Al igual que los en los Array, en los ArrayList se comienza a contar por 0. Hemos comentado anteriormente que los ArrayList son dinámicos por lo que en el ejemplo anterior podríamos añadir un tercer elemento a la lista sin necesidad de crear otra variable nueva, sin embargo, no podríamos hacerlo con el Array.

A continuación vamos a ver como quedaría el ejemplo anterior eliminando la parte relativa a los arrays y utilizando for mejorados para recorrer el ArrayList:

```java
import java.util.ArrayList;
/**
 * @author Pablo Ruiz Soria
 */
public class ClasePrincipal {
    public static void main(String[] args) {
        ArrayList<String> listadeStrings = new ArrayList();
        for(int i = 0; i < 2 ;i++){
            listadeStrings.add("Hello world");
        }
        listadeStrings.remove(0);//eliminamos el primer elemento
        for(String elemento:listadeStrings){
            System.out.println(elemento);
        }
        listadeStrings.add("Nuevo elemento");
        System.out.println("-----------");
        for(String elemento:listadeStrings){
            System.out.println(elemento);
        }
```

```
Output - Modulo2EstructurasDeAlmacenamiento (run)
run:
Hello world
-----------
Hello world
Nuevo elemento
BUILD SUCCESSFUL (total time: 0 seconds)
```

Lo relevante del código anterior lo encontramos en la línea 11 donde hacemos uso del método remove que nos permite borrar el elemento de la lista que nos interese. En la línea 12 y 17 nos encontramos con unos bucles for distintos a los que habíamos utilizado hasta la fecha. En estos bucles for lo que decimos es que extraiga cada vez el siguiente elemento de la lista y lo guarde en una variable llamada elemento de tipo String.

Excepciones

En Java, las excepciones son una clase (documentación) que se lanza cuando ocurre un error en tiempo de ejecución. Es decir, no son errores de compilación de algo que hemos escrito mal y el compilador no comprende sino que son errores que ocurren mientras nuestro programa está siendo usado. Java nos ofrece la posibilidad de controlar estos errores y recuperarnos de ellos sin abortar la ejecución de nuestro programa.

Vamos a ver un ejemplo:

Ejemplo de código con excepciones

Lo que vemos aquí es que el programa compila y se ejecuta sin mayor problema pero durante su ejecución se intenta realizar una división por 0 y se lanza una excepción (ArithmeticException) con el error, nos indica en que línea de nuestro código ocurre (línea 7) y se detiene la ejecución del programa. En un entorno real no sería tolerable que aunque una de las muchas cosas que pueden fallar el programa se detenga, por ello lo que hay que hacer es controlar la excepción y que el programa continúe su ejecución. Vamos a ver como contener el error:

Ejemplo de código con excepción controlada

Lo que hacemos aquí es crear un bloque try-catch-finally. Dentro del try le decimos lo que queremos que haga y si hay algún error de tipo Exception lo capturamos y hacemos lo que está dentro del catch. El bloque finally es opcional y es un trozo de código que se ejecutará siempre falle o no la ejecución en el bloque try. Todas las excepciones, como ArithmeticException, derivan de Exception por eso si en un bloque catch capturamos Exception capturaremos cualquier excepción. Es muy cómodo capturar cualquier excepción en un bloque catch pero lo idea es poner un bloque catch para cada excepción y así saber que falla concretamente. Vamos a ver un ejemplo:

Ejemplo de código con excepción controlada en varios catch

En el código anterior aparecen 2 bloques catch, en el primero de ellos (línea 11) controlo las excepciones que pudiesen ocurrir de tipo aritmético y en el segundo bloque (línea 14) capturo cualquier otro tipo de excepción.

En ocasiones puede resultarnos útil lanzar nosotros mismos un error bien de tiempo genérico (Exception) o bien extendiendo la clase Exception para crearnos nuestro error personalizado. Dado que aún no hemos visto como extender una clase vamos a ver a continuación un trozo de código donde lancemos una excepción genérica:

Ejemplo de código lanzando excepciones

Ejemplo de código lanzando excepciones

En la imagen anterior vemos que creamos una función en la línea 6 que realiza la división de 2 parámetros. En la línea 7 comprobamos que el dividendo no sea 0 ya que no se puede dividir por 0. En caso de que el dividendo sea 0 en la línea 8 lanzamos una excepción (realmente un objeto realmente que inicializamos con su constructor, veremos estos conceptos en el siguiente módulo) con un mensaje personalizado. Como dentro de nuestra función lanzamos una excepción tenemos que indicar de algún modo que este módulo lanza excepciones de tipo Excepcion, por ello en la línea 6 añadimos a nuestra función throws Exception. En el mail lo único que hago es mostrar por pantalla el resultado de llamar a esta función. En el primer caso vemos que entra al catch (línea 15) mientras que en el segundo caso al no lanzarse ninguna excepción no se ejecuta el código contenido en el catch. Ninguno de estos try-catch incluye el bloque finally que como dijimos anteriormente es opcional.

Convenios de escritura

Cuando os hablé de "Constanes y variables" ya comenté algo sobre los convenios de escritura en Java. Vamos a recordarlo:

> Si eres observador/a te habrás dado cuenta de que los tipos básicos empiezan en minúscula mientras que los objetos comienzan en mayúsculas. Es probable que también te hayas percatado de que el nombre que asigno a las variables siempre comienza en minúsculas y el nombre que asigno a las constantes está completamente en mayúsculas, esto son convenios de escritura del lenguaje Java y tu puedes seguir tu propio estilo de código pero es interesante hacer uso del convenio porque facilita la lectura y comprensión del código yen caso de trabajar con otras personas se es mas productivo. En este módulo, un poco mas adelante, hablaremos en mayor profundidad de esta cuestión.
>
> Pablo Ruiz Soria - Capítulo "Constantes y variables"

Hay que dejar claro que un convenio no es mas que eso, un convenio. No estamos obligados a seguirlo pero si va a facilitarnos la labor a la hora de entender código de otras personas. Además, si acostumbramos a nuestro alumnado a seguirlo nos facilitará enormemente la tarea de corrección y detección de errores en su código. Voy a elaborar una pequeña tabla donde recoger algunas generalidades al respecto:

|Elemento|Explicación|Ejemplos| |:--|:--|:--| |Clases|Siempre el primer carácter en mayúsculas. Si son varias palabras, cada una de ellas separada por la primera mayúscula. Buscaremos nombres descriptivos pero no largos. Usaremos nombres en singular.|PreguntaRespuestaClasificacionHistorica| |Variables|Siempre el primer carácter en minúsculas. Si son varias palabras, cada una de ellas separada por la primera mayúscula. Buscaremos nombres descriptivos pero no largos. Usaremos nombres en singular.|preguntarespuestaclasificacionHistorica| |Constantes|Todo en mayúsculas. Si son varias palabras, cada una de ellas separada por guión bajo (_). Buscaremos nombres descriptivos pero no largos. Usaremos nombres en singular.|PREGUNTARESPUESTACLASIFICACION_HISTORICA| |Paquetes|En minúsculas. No es habitual que tengan varias palabras. Si son paquetes para web generalmente usan el dominio invertido.|com.trivinet.modelocom.trivinet.util| |Funciones y métodos|Siempre el primer caracter en minúsculas. Si son varias palabras, cada una de ellas separada por la primera mayúscula. Buscaremos nombres descriptivos pero no largos. Utilizaremos verbos.|obtenerPregunta(...)borrarRespuesta(...)establecerClasificacionHistorica(...)|

|Parámetros|Estableceremos 1 espacio en blanco tras la coma que separa los parámetros en una función para facilitar la lectura.|obtenerPregunta(int id, boolean activa)| |Operadores|Estableceremos 1 espacio en blanco tras los mismos para facilitar la lectura.|a = b + c;a == b| |Paréntesis dentro de paréntesis|Cuando un paréntesis no sea el último procuraremos dejar un espacio en blanco para facilitar la lectura.|escribir(multiplicar(4, 5));| |Bloques de código|Todo el código dentro de un bloque (clase, método, condición, bucle, excepción,...) debe tener la misma identación| |Longitud de las líneas|Evitaremos hacer líneas de longitud mayor a 80 caracteres (en Netbeans viene delimitado por una línea roja). Recordad que una sentencia, en Java, termina con un punto y coma, no con el fin de la línea.|

No me cansaré de insistir en que el compilador no comprueba que sigamos ningún convenio (ni que la lógica del programa sea correcta). El compilador únicamente va a comprobar que sintácticamente el código está escrito del modo adecuado.

Algoritmos y estructuras de resolución de problemas sencillos

Para dar solución a este apartado he preferido crear un proyecto completo con Netbeans de modo que en él dé solución a distintas cuestiones como:

- Cálculo de áreas de cuadrados.
- Cálculo de áreas de rectángulos.
- Cálculo de áreas de circunferencias.
- Cálculo de longitud de circunferencias.
- Dibujo de cuadrados con el símbolo *.
- Dibujo de rectángulos con el símbolo *.
- Dibujo de triángulos con el símbolo *.
- Cálculo de medias aritméticas con vectores.
- Cálculo de medias aritméticas con listas.

Sería demasiado largo y farragoso comentar en este apartado todo el código de este proyecto no obstante he procurado que el código sea muy claro.

Para hacer más sencilla la lectura he dividido el código en 3 clases main. Para ejecutar una u otra basta con que en Netbeans, en las propiedades del proyecto, selecciones que clase main quieres ejecutar en cada momento. También puedes colocarte encima de la clase que contenga main que quieras ejecutar y pulsando sobre ella con el click derecho elegir run.

Puedes descargar el código desde aquí (zip - 0.02 MB)..

Código utilizado en los ejemplos

Módulo 2 Ámbito de las variables (zip - 0.01 MB).

Módulo 2 Cadenas (zip - 0.01 MB).

Módulo 2 Funciones condicionales (zip - 0.01 MB).

Módulo 2 Bucles for (zip - 0.01 MB).

Módulo 2 Bucles while (zip - 0.01 MB).

Módulo 2 Bucles do-while (zip - 0.01 MB).

Módulo 2 Funciones (zip - 0.01 MB).

Módulo 2 Estructuras de almacenamiento de datos (zip - 0.01 MB).

Módulo 2 For mejorado (zip - 0.01 MB).

Módulo 2 Excepciones (zip - 0.01 MB).

Módulo 2 Lanzamiento de excepciones (zip - 0.01 MB).

Módulo 2 Algoritmos y estructuras de resolución de problemas sencillos (zip - 0.02 MB).

Tarea

Tu tarea una vez acabado el segundo módulo consiste en:

- Crear un proyecto llamado Modulo2NombreApellido donde Nombre sea tu nombre y Apellido tu primer apellido. Ejemplo: Modulo2PabloRuiz
- En el proyecto deberás crear una paquete llamado tarea.
- Dentro del paquete tarea deberás crear una clase llamada Principal. En esta clase estará el método main.
- La clase Principal deberá tener las funciones sumarEnteros, restarEnteros, multiplicarEnteros, dividirDecimales, calcularFactorial, sumarArray y sumarLista.
- La función sumarEnteros deberá tener 2 parámetros que serán enteros y deberá mostrar una línea con el resultado de la operación.
- La función restarEnteros deberá tener 2 parámetros que serán enteros y deberá devolver la resta del 2º parámetro al 1º.
- La función multiplicarEnteros deberá tener 2 parámetros que serán enteros y deberá devolver el producto de ambos.
- La función dividirDecimales deberá tener 2 parámetros que serán decimales (double) y devolverá otro decimal (double). Si el divisor es igual a 0 lanzará una excepción genérica con el mensaje "No se puede dividir por cero".
- La función calcularFactorial deberá tener 1 parámetro entero y deberá devolver el factorial del parámetro dado. Puedes elegir entre hacerlo recursivo o no. En caso de que el parámetro sea un número negativo se lanzará una excepción genérica con el mensaje "No puedo calcular el factorial de un número negativo"
- La función sumarArray recibirá un parámetro, que será un vector de enteros. Deberá sumar todos los valores y devolver el resultado.
- La función sumarLista recibirá un parámetro, que será una lista de tipo ArrayList que solo contendrá enteros. Deberá sumar todos y mostrar por pantalla el resultado.
- En el main deberás controlar las excepciones que lancen los métodos para los que hemos establecido excepciones. La forma de controlarlos será mostrando por pantalla el mensaje de la excepción y continuando ejecutando el resto de sentencias.

Introducción a la POO

Introducción a la POO

Por fin tenemos la suficiente base como para comenzar a trabajar con Programación Orientada a Objetos (POO) mas allá de las pequeñas pinceladas que hemos introducido.

Como ya indiqué en el primer módulo la POO es un paradigma de programación. Este paradigma se supone que es una evolución de la programación estructurada con la que quizás estés acostumbrado a trabajar en otros lenguajes de programación. En este paradigma vamos a necesitar un cambio de chip ya que será el objeto quien realice las acciones.

En este tercer módulo del curso vamos familiarizarnos con los principales conceptos de la POO como son la **herencia**, el **polimorfismo** y el **encapsulamiento**.

Objetos

Una vez que hemos hablado de las clases el siguiente paso es hablar de los **objetos**. Los objetos son la clave para entender la programación orientada a objetos. Todo a nuestro alrededor puede ser considerado un objeto.

Consideramos que los objetos de nuestro alrededor tienen 2 características comunes: el **estado** y el **comportamiento**. El estado hace referencia al estado actual de una característica del objeto, pensando en un coche su color, velocidad, marcha,... Mientras que el comportamiento hace referencia a las acciones que el objeto puede llevar a cabo, continuando con el coche: acelerar, frenar, cambiar de marcha,... En ocasiones un objeto puede estar compuesto por otros objetos. Los objetos guardan su estado en campos y exponen su comportamiento a través de métodos. Los métodos son quienes operan con el estado de los objetos con lo que son los encargados de proporcionar la comunicación con el objeto. Al hacer que se acceda a los campos/atributos a través métodos garantizados que en los campos únicamente haya valores que nos interese (lo trataremos en el apartado encapsulación).

Al trabajar con objetos obtenemos una serie de beneficios como:

- Modularidad: Al escribir y mantener el código fuente de cada objeto por separado.
- Ocultación de información: Al hacer únicamente se interacciones con los métodos de un objeto conseguimos ocultar la implementación interna
- Reusabilidad: Cuando trabajemos con la herencia veremos que si tenemos objetos funcionando podemos beneficiarnos de su implementación y evitarnos codificar lo ya hecho.
- Facilidad de uso: Si un determinado objeto nos ocasiona problemas será suficiente con eliminarlo de la aplicación y programar uno que lo reemplace.

Desde un punto de vista técnico, **un objeto es una instancia de una clase**, pero ¿qué significa esto?

Cuando creamos una instancia estamos reservando una zona de memoria dedicada para el objeto en cuestión y, en Java, esa zona va a permanecer ahí mientras la variable en cuestión (u otra) la referencie. Tras leer lo anterior podemos pensar en un objeto como que un puntero a la zona de memoria que ocupa pero en POO no suele hablarse siguiendo esa terminología.

A diferencia de otros lenguajes de programación, como C, **en Java no es necesario liberar las zonas de memoria cuando dejan de utilizarse**. En Java existe un mecanismo llamado **recolector de basura** que se encarga de ir liberando las zonas de memoria que no están siendo referenciadas por ningún objeto.

Con ánimo de aportar otro punto de vista y definiciones vuelvo a recurrir a la wikipedia:

> En el paradigma de programación orientada a objetos (POO, o bien OOP en inglés), un objeto es una unidad dentro de un programa de computadora que **consta de un estado y de un comportamiento**, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución. Un objeto puede ser creado instanciando una clase, como ocurre en la programación orientada a objetos(...)
>
> **Estos objetos interactúan unos con otros**, en contraposición a la visión tradicional en la cual un programa es una colección de subrutinas (funciones o procedimientos), o simplemente una lista de instrucciones para el computador. Cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos de manera similar a un servicio.
>
> En el mundo de la programación orientada a objetos (POO), **un objeto es el resultado de la instanciación de una clase**. Una clase es el anteproyecto que ofrece la funcionalidad en ella definida, pero ésta queda implementada sólo al crear una instancia de la clase, en la forma de un objeto. Por ejemplo: dado un plano para construir sillas (una clase de nombre clase_silla), entonces una silla concreta, en la que podemos sentarnos, construida a partir de este plano, sería un objeto de clase_silla. Es posible crear (construir) múltiples objetos (sillas) utilizando la definición de la clase (plano) anterior. Los conceptos de clase y objetos son análogos a los de tipo de datos y variable(...)
>
> https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)

Clases

Lo primero que vamos a hacer va a ser ver la **definición** que nos ofrece la wikipedia sobre lo que es una clase:

> En informática, una clase es una **plantilla para la creación de objetos** de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. **Cada clase** es un modelo que **define un conjunto de variables** -el estado, **y métodos** apropiados para operar con dichos datos -el comportamiento. Cada objeto creado a partir de la clase se denomina **instancia** de la clase.
>
> Las clases son un pilar fundamental de la programación orientada a objetos. Permiten abstraer los datos y sus operaciones asociadas al modo de una caja negra. Los lenguajes de programación que soportan clases difieren sutilmente en su soporte para diversas características relacionadas con clases. La mayoría soportan diversas formas de **herencia**. Muchos lenguajes también soportan características para proporcionar **encapsulación**, como especificadores de acceso.
>
> https://es.wikipedia.org/wiki/Clase_(inform%C3%A1tica)

En la definición he querido reseñar una serie de ideas clave:

- Plantilla para la creación de objetos
- Define variables y métodos.
- A partir de la clase se crean los objetos. La clase se instancia para crear un objeto.
- Java soporta herencia y encapsulación

Ahora que tenemos una definición de lo que es una clase vamos a ver cual es la **sintaxis** de la misma.

```
NombreClase{
    sentencias del cuerpo de la clase
}
```

Cuando he definido la sintaxis de una clase he indicado que en su interior existen las *sentencias del cuerpo de la clase*, ahora voy a afinar mas. **En Java las clases**, generalmente, **están formadas por una serie de atributos, constructores y métodos**:

- **Atributos**: En caso de existir atributos (variables o constantes) estos pueden ser de tipo primitivo (int, float,...) o puede ser otro objeto (String, Coche, Integer,...).
- **Constructores**: Son métodos especiales que nos van a permitir crear la instancia de nuestra clase. Todas las clases tienen sin excepción al menos 1 constructor, aunque no esté escrito en el programa.
- **Métodos**: Es lo mismo que una función (las veíamos en el módulo 2) pero en este caso reciben este nombre cuando hablamos de objetos.

Como ya indicamos en el módulo anterior, el convenio establece que **la primera letra de una clase debe ser una letra mayúscula**. Personalmente, cuando creo una clase lo hago en el orden que he indicado en la lista anterior. Primero escribo los atributos, a continuación los constructores y después los métodos.

Hasta aquí fácil, ¿verdad? Pues bien, a esta sintaxis básica (que ya habíamos utilizado) le iremos añadiendo una serie de modificadores como public, final, extend y/o implements para ir poco a poco avanzando en nuestros programas.

En Java existe una **jerarquía de clases** en la que **todas las clases derivan de una clase llamada Object**. Si miramos la documentación de cualquier clase veremos que todas, sin excepción, derivan de ella. **Cuando una clase extiende a otra hereda todas las variables y métodos de la superclase (clase de la que hereda)**.

A continuación voy a poner una imagen de la jerarquía de clases de la clase ArrayList según obtenemos de la propia documentación:

Explicación de la documentación

Image not found or type unknown

Lo anterior es una captura de pantalla de la documentación de la clase ArrayList.

- En rojo aparece el paquete al que pertenece la clase
- En amarillo el nombre de la clase
- En verde la jerarquía de clases. ArrayList hereda de AbstractList que a su vez hereda de AbstractCollecion que a su vez hereda de Object. Hablaremos de la herencia mas adelante.
- En azul vemos las interfaces que implementa la clase. Hablaremos de las interfaces mas adelante.
- En morado vemos las clases que son descendiente de la clase ArrayList.

En caso de no poder distinguir los colores, la explicación sigue el orden de los recuadros. Es decir, el primer recuadro es el rojo, el segundo el amarillo,...

Para terminar con este apartado voy a enumerar los distintos **tipos de clases** existentes.

- **Públicas.**
- **Abstractas.**
- **Finales.**

Existe una cuarta opción que es la de no poner modificar a la clase.

Profundizaremos en los tipos de clases en el apartado control de acceso.

Atributos

Anteriormente hemos indicado que una clase puede contener o no atributos (también llamados campos o variables miembro) y que estos atributos podían ser de tipo primitivo o bien otras clases.

Vamos a ver un ejemplo:

Ejemplo de código con atributos

En la clase Clase existe var1 (línea 7) que es un atributo variable de tipo int (tipo primitivo). Existe VAR_2 (línea 8) que es un atributo constante de tipo int (tipo primitivo). Existe var3 (línea 10) que es un atributo variable de la clase Integer y, por último, existe VAR_4 (línea 11) que es un atributo constante de la clase Integer. Aquí he utilizado la clase Integer perteneciente al API de Java, pero si tuviese creadas varias clases podría crear variables de la clase que me interesase. Vamos a ver otro ejemplo:

Ejemplo de código con atributos y composición

En la imagen anterior vemos 2 clases. La clase Motor, que tiene 3 atributos, y la clase Coche, con otros 3 atributos. Si nos fijamos en la línea 7 de la clase Coche veremos que uno de sus atributos es de tipo Motor. En este caso se dice que la clase Coche está compuesta por la clase Motor, no confundir con la herencia (la tratamos mas adelante)

Las clases pueden contener atributos estáticos, para ello antes del tipo utilizaremos la palabra reservada static, pero de esto hablaremos mas adelante.

Constructores

Vamos a ver que **definición** de constructor encontramos en la wikipedia:

> En programación orientada a objetos (POO), un constructor es una subrutina cuya misión es inicializar un objeto de una clase. En el constructor se asignan los valores iniciales del nuevo objeto.
>
> https://es.wikipedia.org/wiki/Constructor_(inform%C3%A1tica)

Es decir, **el constructor nos va a permitir crear una instancia de una clase (un objeto)**. Un constructor es un método especial. Vamos a ver su sintaxis:

```
modificadorDeAcceso NombreDeLaClase(OpcionalmenteParámetros){
    sentencias;
}
```

Voy a ilustrarlo con un ejemplo:

Ejemplo de código con constructores

En la imagen anterior la clase Coche tiene 2 constructores. Un constructor vacío entre las líneas 9 y 11 y un constructor con todos sus parámetros entre las líneas 13 y 17. Por lo que podremos crear objetos de cualquiera de los 2 modos que vemos a continuación:

```
Coche coche = new Coche();
Coche coche2 = new Coche( "Ford", "Grand CMAX", new Motor() );
```

Cuando una clase no tiene escrito ningún constructor el compilador asume que la clase en cuestión tiene el constructor vacío. Es decir, si en el ejemplo anterior suprimimos las líneas 9 a 17 sería equivalente a dejar las líneas 9 a 11 y podríamos crear objetos únicamente así:

```
Coche coche3 = new Coche();
```

Sin embargo, si en el ejemplo anterior suprimimos las líneas 9 a 11 el único constructor que tendrá la clase será el que obliga a dar valor a todos sus atributos, es decir, sólo podríamos utilizar el siguiente modo para crear objetos:

```
Coche coche4 = new Coche( "Ford", "Fiesta", new Motor() );
```

Antes comenté que eran métodos especiales porque si te fijas la sintaxis es igual a la de los métodos con la salvedad de que no se indica el tipo de dato que devuelve y esto es así porque un constructor siempre devuelve como tipo de dato una instancia de la propia clase.

En las líneas 14 a 16 hago uso de la palabra reservada **this**. Cuando en la línea 14 escribo this.marca me estoy refiriendo a la variable marca de la clase y no al parámetro marca. Es decir, cuando escribo this.marca = marca; estoy diciendo que la variable marca de la clase pase a valer lo que valga el parámetro marca. Además de this, en un constructor podemos encontrarnos la palabra reservada **super**, pero el uso de esta lo veremos cuando hablemos de herencia.

En este curso siempre vamos a utilizar el modificador de acceso public para los constructores. Si quieres ver cuando podría tener sentido tener un constructor de acceso privado (private) puedes leer acerca del patrón de diseño Singleton.

Métodos

En el módulo 2 de este curso estuvimos hablando de las funciones. **Un método es** básicamente **lo mismo que una función pero asociado a un objeto**. Cuando en el siguiente apartado de este módulo hablemos de la palabra reservada static espero aclarar mejor esta diferenciación.

La sintaxis es básicamente la misma que en las funciones.

Vamos a ver un ejemplo:

```
1  /**
2   * @author Pablo Ruiz Soria
3   */
4  public class Coche {
5      String marca;
6      String modelo;
7      Motor motor;
8
9      public Coche(){
10
11     }
12
13     public Coche(String marca, String modelo, Motor motor){
14         this.marca = marca;
15         this.modelo = modelo;
16         this.motor = motor;
17     }
18
19     public void escribirInformacion(){
20         System.out.println("Marca: " + marca + ", modelo: " + this.modelo +
21                 ", motor: " + motor);
22     }
23 }
```

En el ejemplo anterior aparece un método entre las líneas 19 y 22. El método es de acceso público (public), no devuelve nada (void), se llama escribirInformacion y no tiene parámetros. El método tiene una única sentencia que ocupa las líneas 20 y 21 y que muestra por pantalla un texto. Fíjate que puedes acceder a los atributos de la clase con o sin this.

En Java, **dentro de una clase puedes tener 2 métodos que se llamen igual siempre y cuando el número de parámetros que utilicen o el tipo varíe**. A continuación tenemos un ejemplo:

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Coche {
    String marca;
    String modelo;
    Motor motor;

    public void escribirInformacion(){
        System.out.println("Marca: " + marca + ", modelo: " + this.modelo +
                ", motor: " + motor);
    }

    public void escribirInformacion(boolean muestraMotor){
        if(muestraMotor){
            escribirInformacion();//puedes anteponer .this
        }else{
            System.out.println("Marca: " + marca + ", "
                    + "modelo: " + this.modelo);
        }
    }
}
```

En el ejemplo anterior vemos que dentro de la misma clase tenemos 2 métodos que se llaman igual (escribirInformacion. Podemos tenerlos porque en la línea 9 definimos uno de ellos sin parámetros y en la línea 14 definimos otro con un solo parámetro de tipo boolean. Podríamos tener otro mas de un solo parámetro siempre y cuando este no fuera de tipo boolean.

En Java existe la [**sobreescritura de métodos**](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica). Esto significa que cuando una clase hereda de otra se puede sobreescribir los métodos de esta. Lo veremos con mas detalle en el apartado de herencia.

Si eres observador/a te habrás dado cuenta de que cuando creo un método no uso la palabra reservada static como hacía con las funciones, en el siguiente capítulo voy a tratar de explicarte el porqué.

Static

Tenemos la posibilidad de crear **variables de clase** y **métodos de clase** haciendo uso de la palabra reservada *static*. Para ello haremos uso de la siguiente sintaxis para las variables de clase:

```
static tipoVariable nombreVariable;
```

Y la siguiente sintaxis para los métodos de clase:

```
static tipoDevuelvo nombreMetodo(parámetros) {
    sentencias;
}
```

En ocasiones nos puede interesar que una variable no esté asociada a un objeto sino que esté asociada a la clase, en ese momento es cuando nos interesa crear una variable de clase. Análogamente puede ocurrir con un método. Si en un momento dado nos interesa que el método esté asociado a la clase y no al objeto entonces debemos hacerlo estático, debemos hacer un método de clase.

Para acceder a una variable de clase debemos hacerlo con la sintaxis

```
NombreDeLaClase.nombreVariable;
```

Y para acceder a un método de clase debemos hacerlo con la sintaxis

```
NombreDeLaClase.nombreMetodo(parámetros);
```

Presta atención a que escribimos el nombre de la clase y no el nombre de ningún objeto.

Creando objetos

Cuando queremos **acceder a un atributo** de una determinada clase debemos hacerlo utilizando la siguiente sintáxis:

```
nombreObjeto.nombreAtributo;
```

Del mismo modo procederemos cuando a lo que queramos **acceder** sea al **método** de un objeto:

```
nombreObjeto.nombreMetodo(parámetros);
```

Lo anterior estará permitido siempre y cuando tengamos el control de acceso adecuado para el atributo o método o cuestión. Sobre el control de acceso os hablaré un poco mas adelante en este módulo, de momento vamos a considerar que no existen estas limitaciones.

Vamos a ver un ejemplo:

Perro.java

```java

/**
 * @author Pablo Ruiz Soria
 */
public class Perro {
    public String nombre;
    public int anioNacimiento;

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

    public void establecerAnioNacimiento(int anioNacimiento){
        this.anioNacimiento = anioNacimiento;
    }

}
```

Pato.java

```java

/**
 * @author Pablo Ruiz Soria
 */
public class Pato {
    public String nombre;
    public int anioNacimiento;

    public Pato(String nombre, int anioNacimiento){
        this.nombre = nombre;
        this.anioNacimiento = anioNacimiento;
    }
}
```

Cliente.java

```java


import java.util.ArrayList;

/**
 * @author Pablo Ruiz Soria
 */
public class Cliente {
    String nombre;
    String pape;//Primer apellido
    String tfnoContacto;
    ArrayList<Perro> perros; //perros que tiene el cliente
    ArrayList<Pato> patos; // patos que tiene el cliente

}
```

Lanzador.java

```java
import java.util.ArrayList;
/**
 * @author Pablo Ruiz Soria
 */
public class Lanzador {
    public static void main(String[] args) {
        Cliente vanesa = new Cliente();

        Perro lassie = new Perro();
        lassie.nombre = "Lassie";
        lassie.establecerAnioNacimiento(1954);//El atributo es privado, no es accesible

        Perro laika = new Perro();
        laika.estableceNombre("Laika");
        laika.anioNacimiento = 1950;

        ArrayList<Perro> perros = new ArrayList();
        perros.add(lassie);
```

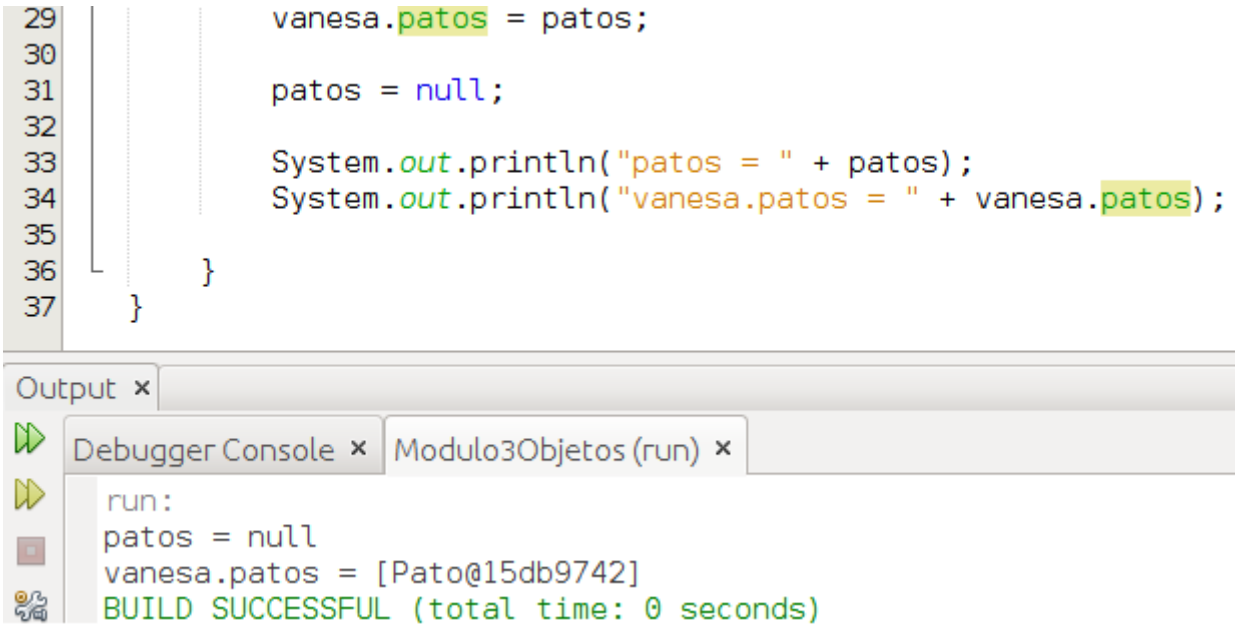
En el ejemplo anterior tenemos 4 clases Perro, Pato, Cliente y Lanzador. Vamos a centrarnos en la clase Lanzador que es la que contiene la función main y es en ella donde ocurre la acción. En las líneas 7, 9, 13, 17, 21 y 23 creamos objetos, fíjate que para ello siempre hacemos uso de la palabra reservada new. En las líneas 10, 15 y 26 a 29 accedemos a los atributos de los objetos en cuestión. En las líneas 11, 14, 18, 19 y 24 accedemos a los métodos de los objetos. Aunque en este ejemplo hayamos accedido a los atributos de este modo no es lo adecuado, generalmente se hará uso del **encapsulamiento**, pero de esto hablaremos mas adelante.

¿Y si sobre el ejemplo anterior quisiésemos acceder al nombre del primer perro de la cliente Vanesa? Pues haríamos lo siguiente:

```
String nombreDelPrimerPerroDeVanesa = vanesa.perros.get(0).nombre;
```

La sentencia anterior es un poco peligrosa y evitaremos utilizarla si antes no hacemos una serie de comprobaciones o incluimos la misma en un bloque try-catch para controlar las excepciones.

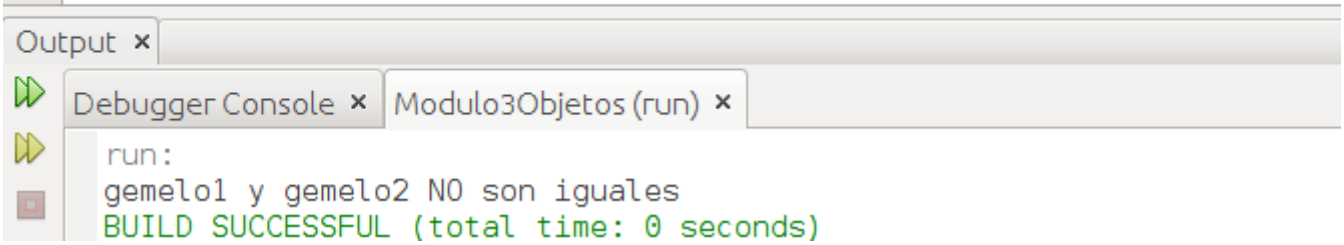
En Lanzador.java vamos a decir que ahora la variable patos valga nulo. Con ella está claro que pasará a valer nulo, pero ¿qué pasará con vanesa.patos que hemos dicho que valga lo que vale esa variable?



Lo que ocurre en la línea 29 es que al atributo patos del objeto vanesa le decimos que pase a apuntar a la misma zona de memoria que apunta la variable patos. Posteriormente decimos en la línea 31 que la variable patos que apuntaba a una determinada zona de memoria pase a no apuntar a ningún sitio (null) pero eso no afecta en que el atributo patos del objeto vanesa cambie su valor. Lo podemos ver en el resultado que aparece en consola ([Pato@15db9742] es la dirección de memoria que la JRE le asigna al objeto)

Otra duda, si creo 2 objetos cuyos atributos tengan exactamente los mismos valores, ¿qué pasará?

```
36              Perro gemelo1 = new Perro();
37              Perro gemelo2 = new Perro();
38              gemelo1.nombre = "Javier";
39              gemelo2.nombre = "Javier";
40
41              if(gemelo1 == gemelo2){
42                  System.out.println("gemelo1 y gemelo2 son iguales");
43              }else{
44                  System.out.println("gemelo1 y gemelo2 NO son iguales");
45              }
46          }
47      }
```



Lo que ocurre es que gemelo1 y gemelo2 apuntan a zonas de memoria distintas por eso el compilador nos dice que los objetos son distintos.

En Java para comparar objetos utilizaremos el método equals. Todos los objetos tienen dicho método puesto que en Java todas las clases derivan de la clase Object y la clase Object tiene este método. Ahora bien, para que equals funcione del modo esperado deberemos sobreescribir el método en nuestra Clase, lo veremos en el apartado de herencia.

Cuando tratamos de acceder a un atributo o método de un objeto y el objeto no apunta a ninguna zona (es nulo) obtendremos una excepción de tipo NullPointerException por lo que antes de acceder a un objeto deberemos realizar las comprobaciones necesarias para evitar este error.

Herencia

Una de las características de un lenguaje de programación orientado a objetos es la **herencia**. En algún capíutlo anterior a este ya hemos visto alguna funcionalidad de la herencia, pero de aquí en adelante vamos a profundizar en ella.

En POO, la herencia es un mecanismo que nos permite extender las funcionalidades de una clase ya existente. De este modo vamos a favorecer la **reutilización** de nuestro código.

La sintaxis cuando queremos heredar de una clase es la siguiente:

```
class nombreClase extends nombreClaseQueQueremosExtender {
```

Hay que indica que **solo podemos heredar de una clase a la vez**. Llamaremos **superclase** a la clase padre y **subclase** a la clase que hereda.

La herencia nos va a acerca a otros conceptos como el **polimorfismo** y la **sobrescritura de métodos** que trataremos a lo largo de este módulo. También va a traer consigo la utilización de nuevas palabras reservadas como **extends** o **super**.

¿Qué ocurre cuando una clase hereda de otra? Pues que la subclase tiene acceso (en función del control de acceso que veremos mas adelante) a los métodos y atributos de la superclase. Vamos a ver un ejemplo a continuación:

Código con ejemplo de herencia

Lo que vemos en el código anterior son 4 clases públicas (Abuelo, Padre, Hijo y Modulo3Clases. La clase Abuelo la definimos en la línea 4 de su código y **al no usar la palabra reservada extends no hereda nada mas que de la clase Object**. En la clase Abuelo definimos una variable y un método. En la línea 5 de la clase Padre indicamos que esta clase extiende a Abuelo con lo que **hereda de dicha clase y puede hacer uso de las variables y médotos** de Abuelo además de los suyos propios. La clase Hijo hereda de Padre (que a su vez heredaba de Abuelo) por lo que en la clase Hijo tendremos acceso tanto a las variables y métodos de Padre como los de Abuelo. En la clase Padre ya hacemos uso de una variable de Abuelo en su línea 8. En la clase Hijo hacemos uso de una varible de Abuelo en la línea 8 y un método de la clase Padre en la línea 9. Por último, tenemos la clase Modulo3Clases que contiene el método main. En las líneas 8 a 10 creamos los objetos de tipo Abuelo, Padre e Hijo. Estos objetos son los que nos dan acceso al código asociado a la clase. En la línea 12 y 13 vemos como desde el objeto padre tenemos acceso a métodos de la clase Abuelo y de la propia clase Padre. Lo mismo ocurre con el objeto hijo en las líneas 18 a 20.

El ejemplo anterior símplemente pretende ilustrar la herencia de atributos y métodos pero ahora vamos a ir un poco mas allá. Vamos a pensar en perros, patos, gatos y vamos a tratar de abstraernos sobre lo que son. Todos ellos son mamíferos y tienen una determinada cantidad de patas. Probablemente todos ellos realicen las mismas acciones. Por ello voy a crear una clase llamada Mamifero con los atributos y métodos comunes y luego crearé una clase para cada animal que contenga sus peculiaridades. Vamos allá:

Código con ejemplos de herencia

Image not found or type unknown

En la clase Mamifero no hay nada reseñable, contiene 1 atributo, 2 constructores y 1 método. La clase Perro en la línea 4 indica que hereda de la clase Mamifero (extends Mamifero), tiene 1 atributo propio y 1 atributo que hereda, tiene 2 constructores y tiene 1 método propio. Dentro de los 2 constructores veo escrito super(4), esto lo que hace es llamar al constructor de su superclase que tiene 1 parámetro de tipo entero, es decir, hace uso del constructor de Mamifero. Dentro del método saludar vemos que existe 1 llamada a un método heredado (getPatas()). La clase Pato tiene 1 atributo propio y 1 atributo que hereda y tiene 1 contrctor y 1 método propio. En el constructor también referencia al constructor de la superclase (con super(2)) y en este caso no hace uso del método que hereda. La clase Gato es prácticamente equivalente a la clase Pato. Si nos fijamos en Lanzador vemos que tenemos la función main y en ella creamos 4 objetos.

En el ejemplo anterior hay que recordar que el hecho de que una clase no indique explícitamente que hereda de ninguna otra implica que dicha clase hereda de Object. Es decir, en el ejemplo anterior Mamifero hereda de Object. Como Perro hereda de Mamifero y Mamifero hereda de Object la clase Perro tendrá acceso a sus propios atributos y métodos y a los de Mamifero y Object.

Vamos a llevar nuestro ejemplo un poco mas allá. Vamos a pensar que quisiésemos organizar un concurso de animales en el cual pudiesen apuntarse un máximo de 100 animales siendo que solo pueden participar perros, patos y gatos ¿haríamos una lista para cada animal? Vamos a ver en el siguiente capítulo como el **polimorfismo** puede ayudarnos.

Polimorfismo

El polimorfismo es la capacidad que nos proporciona un lenguaje de programación orientado a objetos para tratar un objeto como si fuera un objeto de otra clase.

Existen lenguajes de programación donde una variable puede contener prácticamente cualquier tipo de dato, es el caso de los lenguajes PHP, Python y Javascript (recuerda, Javascript no es Java, son lenguajes distintos), mientras que existen otros lenguajes de programación en los que una variable definida de un modo solo pude contener variables de dicho tipo, es el caso de Java. Java es un lenguaje fuertemente tipado.

Debido a este fuerte tipado en ocasiones nos encontramos con que tenemos que moldear un determinado objeto de modo que quepa en el molde de otro, para ello utilizaremos el llamado **casting** o **typecasting**.

Vamos a ver un ejemplo

Ejemplo de código con polimorfismo

El ejemplo anterior no cambia en nada las clases Perro, Pato y Gato utilizadas en el ejemplo visto en el apartado Herencia. Todos los cambios ocurren en la clase Lanzador. Si nos fijamos en ella vemos que en la línea 12 creamos un ArrayList en la cual indicamos que vamos a añadir objetos de la clase Mamifero sin embargo en las líneas 13 y 14 vemos que añadimos 2 objetos de tipo Perro, en la línea 15 un objeto de tipo Pato y en la línea 16 añadimos un objeto de tipo Gato. En este caso el compilador realiza un **casting implícito** ya que todos los tipos anteriores son descendientes de Mamifero. Si en lugar de crear un ArrayList huviesemos creado un ArrayList entonces no habríamos podido insertar al pato ni al gato. Prosigamos, en la línea 18 nos encontramos con un bucle for mejorado que recorreaá todos los elementos del array y los guardará en una variable llamada animal de tipo Mamifero. Vemos que en las líneas 19, 21 y 23 usamos una palabra reservada nueva (instanceof), esta palabra lo que hace es devolver verdadero en caso de que la variable animal sea una instancia de Perro, Pato o Gato respectivamente. Cuando la condición es cierta entramos dentro del if y se realiza un **casting explícito**, es decir, le decimos que comportamiento queremos que tenga ese objeto para que así podamos acceder a sus métodos y atributos. En la línea 20 nos encontramos ((Perro) animal).saluda(); aquí lo que ocurre es que se realiza el cast sobre animal para que se comporte como si fuera de tipo Perro y una vez que se ha realizado el cast y ya no es de tipo animal sino que es de tipo Perro, entonces es cuando tenemos acceso al método saludar y no antes. Ocurre lo mismo en las líneas 22 y 24.

En nuestro ejemplo de herencia y polimorfismo hemos creado una clase (Mamifero) que queríamos sirviese de base para otras (Perro, Pato y Gato) pero que no hemos llegado a utilizar. En el capítulo

Clases abstractas e interfaces veremos cual habría sido la forma mas adecuada de diseñar esta jerarquía de clases.

Pregunta Verdadero-Falso

Java y Javascript son el mismo lenguaje de programación. Nos referimos a Javascript como Java para tener un lenguaje mas fluido.

VerdaderoFalsoJava y Javascript son lenguajes de programación distintos. Su sintaxis es parecida al igual que ocurre entre C# y Java o entre C++ y Java.

En el ejemplo anterior, sería correcto escribir Gato gato = (Gato) nala;

VerdaderoFalsoNo podría hacerse lo que se indica en la pregunta ya que nala es un objeto de tipo Perro que hereda de Mamifero y Gato hereda de Mamifero. En la jerarquía de clases derivan de la clase Mamifero pero a partir de ese momento son ramas distintas de la jerarquía. Si podría hacer Mamifero mamifero = (Mamifero) nala;

Sobreescritura de métodos

Otra característica asociada a la herencia es la **sobreescritura de métodos**. La wikipedia se refiere a ella como [redefinición de métodos](https://es.wikipedia.org/wiki/Herencia_(inform%C3%A1tica) pero yo nunca la he visto denominada de ese modo con anterioridad.

Cuando dentro del apartado Clases estuvimos hablando de los métodos nombramos por primera vez la sobreescritura de métodos. No hay que confundir la sobreescritura de métodos con que un mismo método pueda ser definido de modos distintos.

La sobreescritura de métodos nos permite redifinir un método que heredamos para que este funcione de acuerdo a nuestras necesidades y no a lo definido en la superclase. Cuando en un objeto llamamos a un método el compilador comprueba si el método existe en nuestro objeto, si existe lo usa y si no existe en nuestro objeto entonces lo busca en la superclase. Esto ocurre así hasta que el compilador encuentra el método definido. El compilador busca el método de abajo a arriba.

Vamos a ver un ejemplo

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Persona {
    int anioNac;
    String nombre;
    String pape; //1er apellido

    public Persona(int anioNac, String nombre, String pape){
        this.anioNac = anioNac;
        this.nombre = nombre;
        this.pape = pape;
    }

    @Override
    public String toString(){
        return "Me llamo " + nombre + " " + pape
                + " y nací en " + anioNac + ".";
    }
}
```

```java
/**
 * @author Pablo Ruiz Soria
 */
public class Animal {
    String nombre;
    int patas;

    public Animal(String nombre, int patas){
        this.nombre = nombre;
        this.patas = patas;
    }
}
```

```java
/**
 * @author Pablo Ruiz Soria
 */
public class NewMain {
    public static void main(String[] args) {
        Persona aldara = new Persona(2016, "Aldara", "Ruiz");
        Persona pablo = new Persona(1983, "Pablo", "Ruiz");
        Animal nala = new Animal("Nala", 4);
        Animal donald = new Animal("Donald", 2);

        System.out.println(aldara.toString());
        System.out.println(donald.toString());
        System.out.println(nala.toString());
        System.out.println(pablo.toString());
    }
}
```

Output ×

Debugger Console × | Modulo3SobreescrituraDeMetodos (run) ×

```
run:
Me llamo Aldara Ruiz y nací en 2016.
Animal@15db9742
Animal@6d06d69c
Me llamo Pablo Ruiz y nací en 1983.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Por simplicidad he decidido escribir 2 clases (Persona y Animal) que al no usar la palabra reservada extended derivan implicitamente de la clase Object. Si miramos la documentación de Object veremos que tiene un método llamado toString. Este método lo que hace es indicar la dirección de memoria en que el compilador ha guardado el objeto. En la clase Persona en las líneas 15 a 19 he redefinido el método toString dándole un comportamiento distinto mientras que en la clase Animal no lo he redefinido. Si volvemos a la clase Persona vemos que en la línea 15 aparece una anotación que le indica que estamos sobreescribiendo el método, esta anotación no es necesaria pero en teoría agiliza la compilación y ejecución de nuestros programas. Si vamos a la función main vemos que en las líneas 6 a 9 creamos los objetos y en las líneas 11 a 14 llamamos al método toString de dichos objetos. En el caso de los objetos de tipo persona se ejecuta el método que hemos redefinido mientras que en los objetos de tipo animal se utiliza el método de la superclase.

Cuando redefinimos un métodos podemos hacer uso del propio método que estamos redefiniendo, para ello haremos uso de la palabra reservada super. Voy a modificar el código del ejemplo anterior para que podamos verlo:

```
16      public String toString(){
17          return "Me llamo " + nombre + " " + pape
                    + " y nací en " + anioNac +
                    ". El compilador me ubica en " + super.toString();
19      }
```

```
Debugger Console × | Modulo3SobreescrituraDeMetodos (run) ×
run:
Me llamo Aldara Ruiz y nací en 2016. El compilador me ubica en Persona@15db9742
Animal@6d06d69c
Animal@7852e922
Me llamo Pablo Ruiz y nací en 1983. El compilador me ubica en Persona@4e25154f
BUILD SUCCESSFUL (total time: 0 seconds)
```

En el extracto anterior vemos que he añadido super.toString() para decirle a mi método toString que llame al método toString de la superclase. Debajo del código aparece el resultado que se produce al ejecutar el programa tras el cambio.

Control de acceso

Hasta el momento hemos visto que desde cualquier clase se puede acceder a cualquier otra clase y a cualquier atributo o método sin ningún control. Esto lo hemos hecho de este modo para facilitar la comprensión de los distintos conceptos tratados a lo largo del curso y de este modo no distraernos con cuestiones de control de acceso pero a partir de este momento vamos a hablar del **control de acceso** en clases, atributos, constructores y métodos para así empezar a desarrollar nuestros programas de un modo mas adecuado.

a clases

Cuando definimos una clase podemos hacerlo con la palabra reservada public, abstract, final o ninguna de los anteriores. Existe alguna otra posibilidad pero no entraremos en ella.

|Clase|Descripción| |:--|:--| |public|Una clase pública es accesible desde cualquier clase. Para ser utilizada desde otro paquete debe ser importada.| |abstract|Una clase abstracta es una clase destinada a que se herede de ella. No puede ser instanciada. Debe tener al menos un método abstracto.| |final|Una clase final es aquella que no permite que se herede de ella.|

Así podemos combinar las distintas palabras clave:

```
public class Ejemplo1{

class Ejemplo2{

public abstract class Ejemplo3{

abstract class Ejemplo4{

public final class Ejemplo5{

final class Ejemplo6{
```

En las 6 líneas anteriores tenemos todas las combinaciones con las que vamos a trabajar en el curso.

La clase Ejemplo1 es una clase pública por lo que será accesible desde cualquier paquete y se podrán crear instancias de ella y usarla como superclase.

La clase Ejemplo2 será accesible únicamente desde el paquete en que sea definida y se podrán crear instancias de ella y usarla como superclase.

La clase Ejemplo3 es una clase pública por lo que será accesible desde cualquier paquete y al ser abstracta no se podrán crear instancias de ella, deberá contener al menos un método abstracto y se podrá usar como superclase.

La clase Ejemplo4 será accesible únicamente desde el paquete en que sea definida y al ser abstracta no se podrán crear instancias de ella, deberá contener al menos un método abstracto y se podrá usar como superclase.

La clase Ejemplo5 es una clase pública por lo que será accesible desde cualquier paquete pudiendo ser instanciada pero no usada como superclase.

La clase ejemplo6 será accesible únicamente desde el paquete en que sea definida pudiendo ser instanciada pero no usada como superclase.

a atributos

Existen las siguientes palabras clave cuando queremos controla el acceso a las variables miembro de nuestras clases. Vamos a verlas:

|Atributo|Descripción| |:--|:--| |public|El campo es accesible desde todas las clases| |private|El campo es accesible únicamente desde la propia clase| |final|El campo no puede ser modificado y al definirse debe establecerse su valor.|

Vamos a ver algún ejemplo:

public String ejemplo1;

String ejemplo2;

private String ejemplo3;

public final String ejemplo4 = "trivinet.com";

final String ejemplo5 = "recurso didáctico gratuito";

private final String ejemplo6 = "de alta calidad";

Las 6 combinaciones anteriores serían todas las posibles.

ejemplo1 sería accesible y modificable desde cualquier clase.

ejemplo2 sería accesible únicamente desde las clases que pertenezcan al mismo paquete y cualquiera de ellas podría modificarla.

ejemplo3 sería accesible únicamente desde la clase en que se ha definido y podría ser modificada.

ejemplo4 sería accesible desde cualquier clase. No se podría modificar su valor.

ejemplo5 sería accesible únicamente desde las clases que pertenezcan al mismo paquete. No se podría modificar su valor.

ejemplo6 sería accesible únicamente desde la clase en que se ha definido. No se podría modificar su valor.

Es común que los campos se definan como privados para así sacar todo el potencial de la **encapsulación** (la veremos en el siguiente capítulo).

a constructores

|Constructor|Descripción| |:--|:--| |public|Cualquier clase puede crear instancias de la clase que contenga un constructor público.| |protected|Solamente las subclases de la clase que contiene un constructor protegido pueden crear instancias de la clase.| |private|Ninguna clase puede crear instancias de una clase que tiene un constructor privado excepto la propia clase a través de alguno de sus métodos públicos.|

El patrón de diseño singleton es un ejemplo a través del cual ver la utilidad de definir un constructor privado.

a métodos

A la hora de establecer el control de acceso a un método en el curso vamos a trabajar con public, private, protected, abstract y final.

|Método|Descripción| |:--|:--| |public|El método es accesible desde la propia clase, el paquete al que pertenezca la clase, las subclases y el resto del mundo.| |protected|El método es accesible desde la propia clase, el paquete al que pertenezca la clase y las subclases.| |private|El método es accesible desde la propia clase.| |abstract|El método no está definido en esta clase sino que será definido en la clase que herede de la clase que contenga este método.| |final|El método no puede ser sobreescrito.|

Cuando un método es abstracto no se pone el bloque de código que va entre las llaves ({}) ni las propias llaves, esto tiene sentido porque el método en cuestión debe definirse en la clase que herede la clase que contenga el método abstracto.

Vamos a ver algunas combinaciones:

public void ejemplo1(){

protected void ejemplo2(){

void ejemplo3(){

private void ejemplo4(){

public abstract void ejemplo5();

protected abstract void ejemplo6();

abstract void ejemplo7();

public final void ejemplo8(){

protected final void ejemplo9(){

final void ejemplo10(){

private final void ejemplo11(){

En los 11 ejemplos anteriores no hay valor de retorno (void) y como puede apreciarse no existe la combinación private abstract ya que no tendría sentido. Existe alguna otra palabra reservada que da lugar a mas combinaciones pero estas son suficientes para el objetivo del curso.

En la documentación oficial de Java sobre el control de acceso podemos encontrar mas información (en inglés).

Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos y consiste en ocultar el estado interno del objeto y obligar a que toda interacción se realice a través de los métodos del objeto.

En el código que hemos desarrollado hasta el momento hemos dado libre acceso a los atributos de nuestros objetos lo cual presenta riesgos ya que no tenemos ningún tipo de control sobre lo que puede o no hacerse con ellos. Nos va a interesar que a nuestros atributos solo se pueda acceder a través de los métodos, de modo que obtengamos control sobre lo que puede hacerse con los atributos.

Para ello, el acceso a los atributos se establece como privado y se crean 2 métodos por cada atributo, un **getter** y un **setter**. El getter de un atributo se llamará getNombreAtributo mientras que el setter de un atributo se llamará setNombreAtributo. Vamos a ver un ejemplo para entender esto mejor:

Código con ejemplo de encapsulamiento

Si nos fijamos en Mesa.java veremos que es una clase con 1 atributo privado llamado color y 2 métodos que se llaman getColor y setColor. Los métodos anteriores serían, respectivamente, el getter y setter de del atributo color. Ahora que ya conocemos el control de acceso que sobre clases, atributos, constructores y métodos puede realizarse sabemos que no puede accederse directamente al atributo ni para obtener su valor ni para establecer su valor, siempre que quieran realizarse esas acciones habrá que pasar respectivamente por el getter y el setter y en ellos podremos programar aquello que nos interese. Si nos fijamos en Principal.java veremos que si ahora queremos acceder directamente al atributo para modificar u obtener su valor no podemos (he dejado comentado el código que falla) y que para hacerlo estamos obligados a utilizar los getter y setters que hemos creado antes.

Si bien es cierto que podríamos llamar a los getters y setters de cualquier otro modo **por convenio se utiliza la sintaxis que hemos visto con anterioridad**. Hacer uso de este convenio nos facilitará trabajar con el resto del mundo y nos permitirá ampliar las capacidades de nuestro código utilizando frameworks existentes que hacen uso del convenio y que si no seguimos no podremos utilizar.

¿Aún no les ves sentido?

Voy a modificar ligeramente el código de Mesa.java para tratar de crear un ejemplo mejor:

Ejemplo de código con encapsulamiento desarrollado

A la clase Mesa le he añadido un nuevo atributo llamado numeroDeVecesPintada en el cual voy a almacenar el número de veces que he cambiado el color de la mesa. Además, ahora la mesa quiero que solo pueda pintarse de 3 colores distintos. El número de veces que se pintado la mesa no quiero que pueda ser cambiado en cualquier momento, solo quiero que se autoincremente en 1 cuando realmente se pinte la mesa. Por ello en el setter de color hago que solo cambie el valor del atributo cuando recibo un color válido y si eso ocurre además aumento en 1 las veces que he pintado la mesa. Si en el setter de color (setColor) no recibo un valor válido entonces no cambio el atributo color de la mesa ni incremento en 1 las veces que se ha pintado la mesa. Como además no quiero que nadie pueda modificar el valor de las veces que se pinta la mesa no hago un setter para este atributo.

Interfaces y clases abstractas

Ya hemos hablado con anterioridad de las **clases abstractas** (capítulo polimorfismo), en este capítulo profundizaremos un poco mas sobre ellas y además hablaremos por primera vez de las **interfaces**.

Interfaces

Cuando alguien te pregunta que es una interface suele decirse que es un contrato, una forma de asegurarse que todo el mundo que implemente una interface va a hacer algo. No nos importa como vayan a hacerlo pero estamos seguros que si una clase implemente una interface entonces va a tener que cumplir el contrato e implementar lo acordado.

Vamos a ver la sintaxis

```
public interface NombreInterface{
    sentencias;
}
```

Y cuando una clase implementa una interface la sintaxis es

```
class NombreClase implements NombreInterface{
    sentencias;
}
```

Una clase puede implementar 0 o varias interfaces al mismo tiempo. En caso de implementar varias interfaces separaremos sus nombres por comas. El hecho de poder implementar varias interfaces al mismo tiempo nos sirve para esquivar la limitación de que una clase solo pueda heredar de una única clase. En caso de que una clase extendise a otra clase y a la vez implementase 1 o varias interfaces la sintaxis sería la siguiente:

```
class NombreClase extends SuperClase implements NombreInterface1, nombreInterface2,...{
    sentencias;
```

```
}
```

Las interfaces pueden ser públicas (accesibles desde cualquier lugar) o sin modificador de acceso (accesibles desde el paquete de la interface).

Una interface no puede ser instanciada (si se podría pero es demasiado avanzado para el curso).

Una interface puede contener métodos abstractos y métodos estáticos, ambos se considerarán públicos (aunque no se indique). También pueden añadir métodos por defecto (con el modificador default) pero no los vamos a ver en este curso.

Una interface puede contener atributos que a todos los efectos será una constante, estaremos obligados a darle valor. Los atributos que creemos se considerarán públicos, estáticos y finales por lo que podemos omitir los modificadores.

Una interface puede extender otras inferfaces, lo hará con la siguiente sintaxis

```
public interface NombreInterface extends NombreOtraInterface1, NombreOtraInterface2,...{
    sentencias;
}
```

Si, es correcto extends, no me he equivocado.

Clases abstractas

La sintaxis necesaria para crear una clase abstracta es la siguiente

```
abstract class NombreClase{
    sentencias;
}
```

Las clases abstractas pueden incluir o no **métodos abstractos**. Las clases abstractas no pueden ser instanciadas pero si pueden ser usadas como subclases.

Un método abstracto es un método que está declarado pero no está implementado. En su sintaxis se suprimen las llaves ({}) y tras cerrar el paántesis que contiene los parámentros se pone un punto y coma.

```
public abstract void montar();
```

Si una clase contiene un método abstracto tiene que ser obligatoriamente una clase abstracta.

Cuando una clase hereda de una clase abstracta y la superclase contiene un método abstracto estamos obligados a implementarlo.

Diferencias entre clases abstractas e interfaces

Las clases abstractas y las interfaces son similares. No puedes instanciarlas y pueden contener métodos implementados o no.

En las clases abstractas puedes disponer de atributos sin que estos sean constantes (como ocurre en las interfaces) y además puedes crear métodos públicos, protegidos o privados (en las interfaces todos eran públicos).

Solo puedes extender una clase (abstracta o no) mientas que puedes implementar cualquier número de interfaces.

Entonces, ¿cuándo elijo una u otra?

- Clases abstractas cuando se cunpla alguna de las condiciones siguientes:
  - Quieres compartir código entre muchas clases relacionadas
  - Esperas que las clases que extiendan tu clase abstracta tengan en común métodos o campos
  - Quieres disponer de atributos no estáticos o no finales. Esto te habilita para definir métodos que puedan acceder y modificar el estado del objeto al que pertenecen
- Interfaces cuando se cumpla alguna de las condiciones posteriores:
  - Esperas que clases sin relación entre si implementen tu interface.
  - Quieres un comportamiento específico sin importante la implementación
  - Quieres disponer de herencia múltiple.

Si deseas ampliar la información, en la documentación oficial de Java existe un tutorial específico sobre interfaces y herencia (en inglés).

En el apartado "código utilizado en los ejemplos" dejo un proyecto en el cual hago uso de interfaces y clases abstractas.

Organización del código: Paquetes

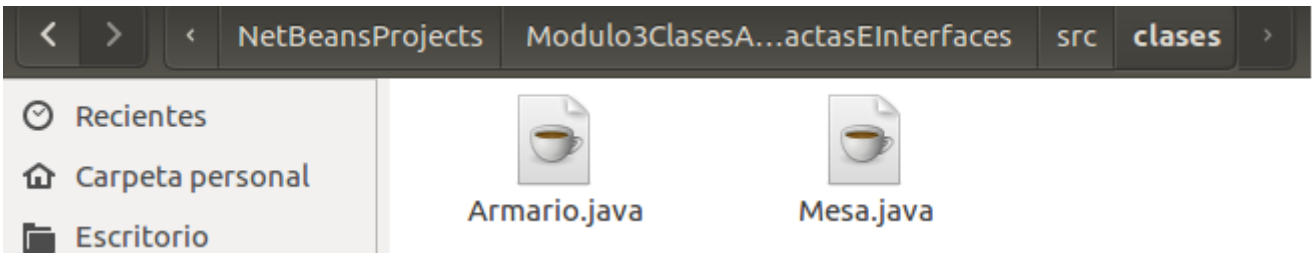
Un paquete es una agrupación lógica de clases o interfaces relacionadas. La creación de paquetes nos permite dotar a nuestro código de mayor protección a través del control de acceso y además nos permite facilitar la organización de nuestros programas. Si pensamos en los ficheros con los que trabajamos a diario no los dejamos todos sueltos en el escritorio sino que creamos una serie de carpetas/directorios donde los agrupamos de modo lógico con el fin de facilitar nuestra organización.

El convenio dice que los paquetes deben nombrarse en minúsculas y cuando se necesitan usar varias palabras separarlas por un guión bajo. Además, suele utilizarse el dominio de nuestra empresa invertido, de este modo, podemos diferenciar clases que compartan nombre.

Para indicar que una clase pertenece a un determinado paquete debemos indicarlo usando package nombreDelPaquete y ubicando la clase dentro de dicha ruta de carpetas. Vamos a ver un ejemplo:

```
1  package clases;
2
3  import clases_abstractas.Mueble;
4  import interfaces.Transporte;
5
6  /**
7   * @author Pablo Ruiz Soria
8   */
9  public class Armario extends Mueble implements Transporte{
```

Por lo que la clase Armario deberá estar ubicada en la carpeta clases

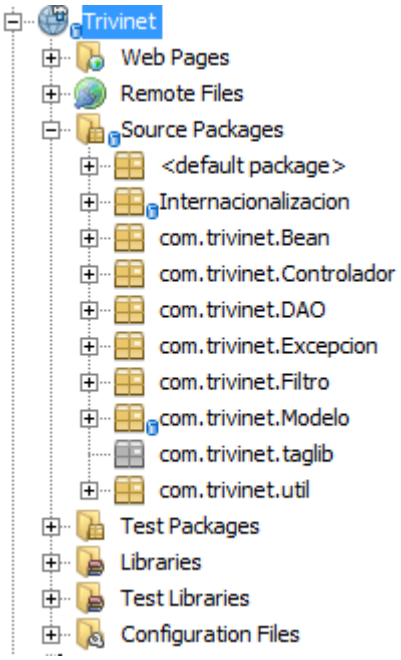


Cuando en nuestros programas queremos usar una clase que no pertenece a nuestro paquete (y tenemos permiso para ello) existen 2 posibilidades:

- Importar la clase, tal como vemos en las líneas 3 y 4 de la imagen anterior

- Referenciarla completamente cada vez que vayamos a usarla. Eliminaríamos las líneas 3 y 4 y cada vez que quisiesemos hacer referencia a Mueble escribiríamos clases_abstractas.Mueble y cada vez que quisiésemos hacer referencia a Transporte escribiríamos interfaces.Transporte

A continuación vamos a ver la organización en paquetes de un proyecto real, el proyecto a partir del cual se desarrolla el recurso didáctico trivinet.com



El proyecto de la imagen está formado por 80 clases agrupadas en 9 paquetes distintos en función de las funcionalidades que cada clase tiene.

Java nos proporciona una cantidad de paquetes enormes lista para utilizar en nuestros paquetes, esta librería recibe el nombre de API (Application Programming Interface) La especificación completa del API de Java 8 para su edición estándar está accesible aquí.

Código utilizado en los ejemplos

Módulo 3 Clases (zip - 0.01 MB).

Módulo 3 Atributos (zip - 0.01 MB).

Módulo 3 Objetos (zip - 0.01 MB).

Módulo 3 Herencia (zip - 0.01 MB).

Módulo 3 Polimorfismo (zip - 0.02 MB).

Módulo 3 Sobreescritura de métodos (zip - 0.01 MB).

Módulo 3 Control de acceso (zip - 0.01 MB).

Módulo 3 Encapsulamiento (zip - 0.01 MB).

Módulo 3 Interfaces y clases abstractas (zip - 0.01 MB).

Tarea

Tu tarea una vez acabado el tercer módulo consiste en:

- Crear un proyecto llamado Modulo3NombreApellido donde Nombre sea tu nombre y Apellido tu primer apellido. Ejemplo: Modulo3PabloRuiz
- En el proyecto deberá estar organizado por paquetes de modo que tengas, al menos, los paquetes clase, clase_abstracta, interf y lanzador.
- Dentro del paquete lanzador deberás crear una clase llamada Principal. En esta clase estará el método main.
- Vamos a modelizar un tienda de bicicletas.
  - Tendrás una clase abstracta llamada Bicicleta
  - Tendrás unas subclases de Bicicleta llamadas BiciMontania, BiciPaseo, Tandem. Estas subclases deberán sobreescribir al método toString de Object haciendo que digan de que tipo son, su color y precio.
  - Todas las bicicletas tendrán dos campos comunes color (String) y precio (double).
  - Quiero que todas las bicicletas implementen el método pintar, que lo que hará será cambiar el color pero me interesa que este método pintar también se pueda aplicar a otras posibles modelizaciones futuras que pudiera hacer, no debe ser exclusivo de las bicicletas. Además quiero que el coste de pintar (lo que sea) sea fijo y sea 90.
  - Las bicicletas de montaña deberán almacenar la marcha que tiene la bici (pudiendo tomar valores únicamente entre 1 y 6).
  - Las bicicletas de paseo deberán almacenar la velocidad a la que se circula.
  - Los tandem deberán almacenar el número de asientos (pudiendo tomar únicamente los valores 2 y 3).
  - Todos los campos de todas las clases deberán seguir los principios de encapsulamiento.
  - La clase BiciMontania deberá tener un único constructor a través del cual se establezca el valor del color, precio y marcha.
  - La clase BiciPaseo deberá tener 2 constructores, uno a través del cual se establezca el valor del color, precio y velocidad y otro a través del cual se establezca el color y precio.
  - La clase Tandem deberá tener un único constructor a través del cual se establezca el valor del color, precio y número de asientos.
  - Los constructores deberán controlar las condiciones existentes.
  - La BiciMontania tendrá un método llamado aumentarMarcha que no tendrá parámetros y aumentará en 1 la marcha actual (siempre que cumpla las condiciones anteriores). También tendrá un método llamado disminuirMarcha que no tendrá parámetros y reducirá en 1 la marcha actual (siempre que cumpla las condiciones anteriores). No se podrá modificar la marcha desde ningún otro lugar.

- En la clase Principal deberás crear una lista con todas las bicicletas que tengas en la tienda. En este momento tienes 2 bicicletas de montaña, 1 bici de paseo y 2 tandems. Crea los objetos y añádelos a la lista. Posteriormente interacciona con ellos y al final recorre la lista para llamar al método toString de cada objeto.

Terminando

Terminando

El objetivo de este último módulo es facilitarte mas recursos. Por ello voy a realizar 1 ejemplo completo desde su fase de análisis hasta su fase de codificación tratando de explicarte aquellas partes mas relevantes. También voy a ver unas pinceladas sobre como crear una interfaz gráfica de usuario, facilitarte algo de bibliografía y, como siempre, dejarte todo el código completo y una tarea.

Taller con POO

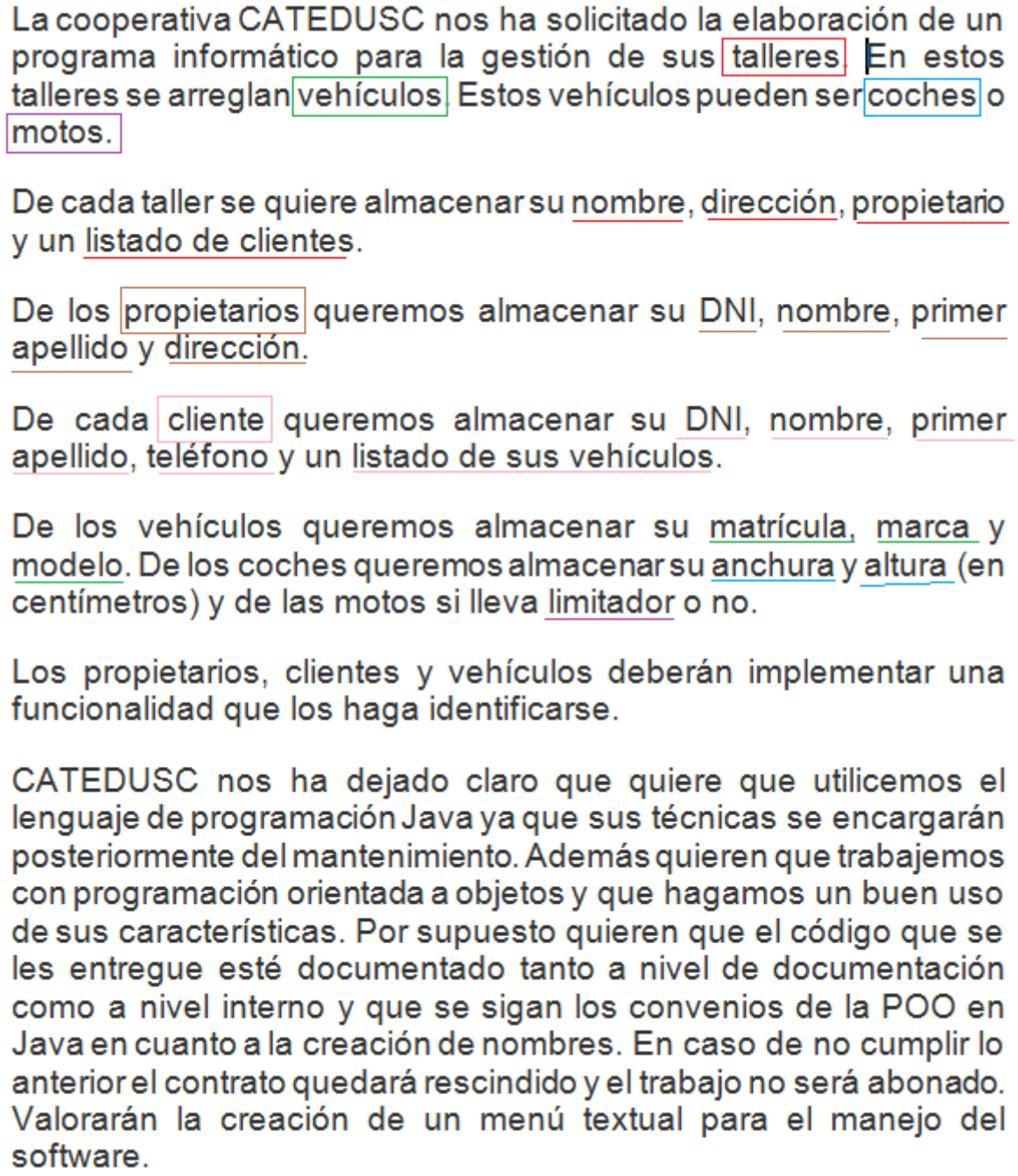
Antes de ponernos a hacer nada vamos a definir nuestros problema.

> La cooperativa CATEDUSC nos ha solicitado la elaboración de un programa informático para la gestión de sus talleres. En estos talleres se arreglan vehículos. Estos vehículos pueden ser coches o motos.
>
> De cada taller se quiere almacenar su nombre, dirección, propietario y un listado de clientes.
>
> De los propietarios queremos almacenar su DNI, nombre, primer apellido y dirección.
>
> De cada cliente queremos almacenar su DNI, nombre, primer apellido, teléfono y un listado de sus vehículos.
>
> De los vehículos queremos almacenar su matrícula, marca y modelo. De los coches queremos almacenar su anchura y altura (en centímetros) y de las motos si lleva limitador o no.
>
> Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.
>
> CATEDUSC nos ha dejado claro que quiere que utilicemos el lenguaje de programación Java ya que sus técnicas se encargarán posteriormente del mantenimiento. Además quieren que trabajemos con programación orientada a objetos y que hagamos un buen uso de sus características. Por supuesto quieren que el código que se les entregue esté documentado tanto a nivel de documentación como a nivel interno y que se sigan los convenios de la POO en Java en cuanto a la creación de nombres. En caso de no cumplir lo anterior el contrato quedará rescindido y el trabajo no será abonado. Valorarán la creación de un menú textual para el manejo del software.

Ahora que tenemos definido nuestro programa es el momento de ponernos a pensar antes de empezar a codificar. Para ello en los siguientes capítulos empezaremos por definir las clases que hemos detectado y las relaciones existentes entre ellas. Una vez que tengamos definido lo anterior será el momento de empezar a escribir el código fuente.

Identificación de las clases

Mi consejo para los primeros programas en los que vayáis a trabajar con programación orientada a objetos es que os imprimais el enunciado del problema y con un lápiz pongáis un recuadro sobre aquellos elementos que definái como un objeto y subrayéis aquellos elementos que identifiquéis como atributos del mismo.



La cooperativa CATEDUSC nos ha solicitado la elaboración de un programa informático para la gestión de sus talleres. En estos talleres se arreglan vehículos. Estos vehículos pueden ser coches o motos.

De cada taller se quiere almacenar su nombre, dirección, propietario y un listado de clientes.

De los propietarios queremos almacenar su DNI, nombre, primer apellido y dirección.

De cada cliente queremos almacenar su DNI, nombre, primer apellido, teléfono y un listado de sus vehículos.

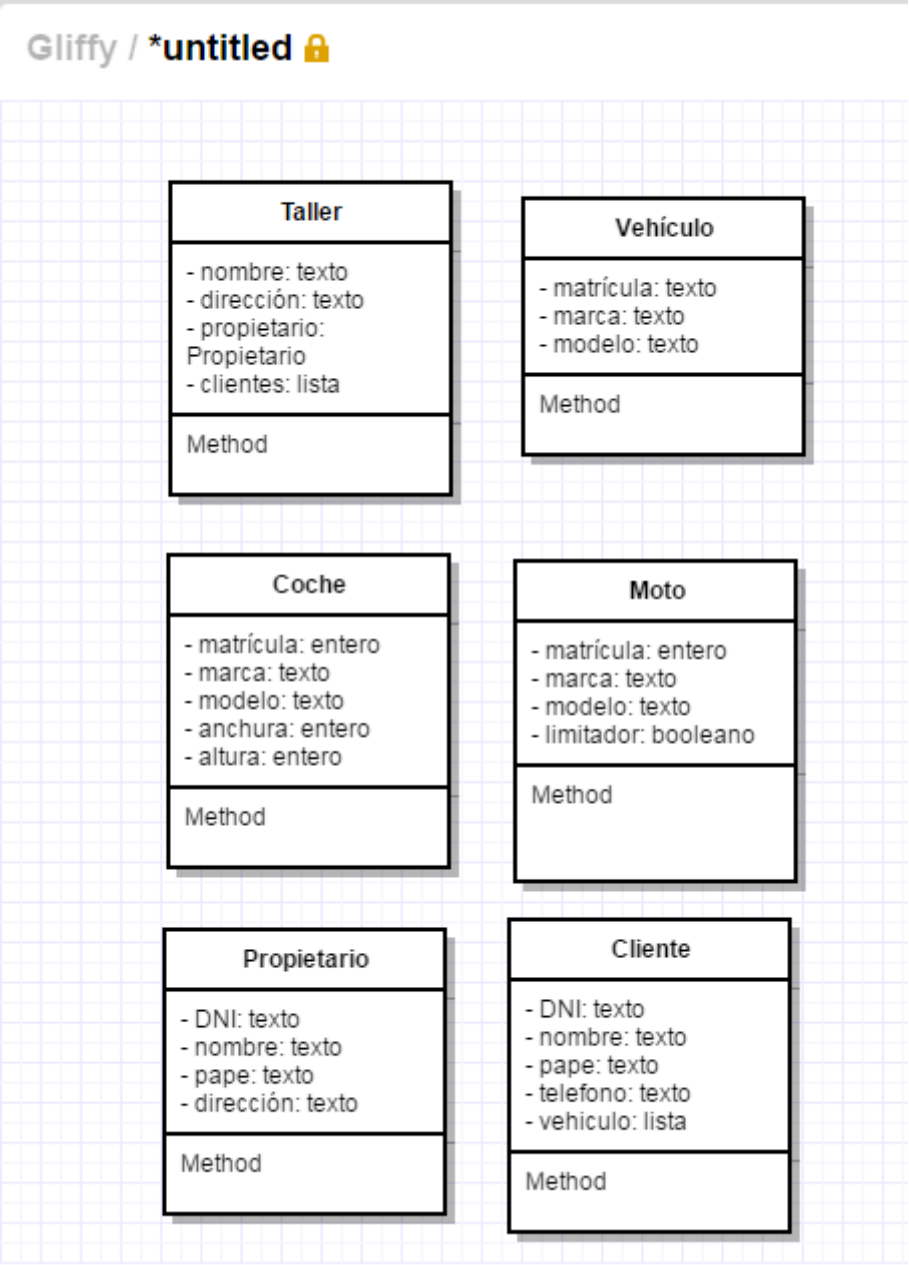
De los vehículos queremos almacenar su matrícula, marca y modelo. De los coches queremos almacenar su anchura y altura (en centímetros) y de las motos si lleva limitador o no.

Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.

CATEDUSC nos ha dejado claro que quiere que utilicemos el lenguaje de programación Java ya que sus técnicas se encargarán posteriormente del mantenimiento. Además quieren que trabajemos con programación orientada a objetos y que hagamos un buen uso de sus características. Por supuesto quieren que el código que se les entregue esté documentado tanto a nivel de documentación como a nivel interno y que se sigan los convenios de la POO en Java en cuanto a la creación de nombres. En caso de no cumplir lo anterior el contrato quedará rescindido y el trabajo no será abonado. Valorarán la creación de un menú textual para el manejo del software.

En la imagen superior vemos que identifico las clases Taller, Vehículo, Coche, Moto, Propietario y Cliente y una serie de atributos para clase. He subrayado del mismo color los atributos que corresponden a cada clase, la cual he recuadrado en ese mismo color.

Una vez hemos identificado las clases y sus atributos lo ideal sería utilizar algún modo estandar de representarlo. Yo he optado por utilizar un lenguaje unificado de modelado (UML) para esta tarea y concretamente lo relativo a los diagramas de clases que es la tarea que nos ocupa. Con UML cada clase estará contenida en un recuadro que dividiré horizontalmente en 3 recuadros. En el recuadro superior escribiré el nombre de la clase, en el recuadro inferior los atributos de la clase y en el recuadro inferior los métodos de la clase. Vamos a ver como quedarían nuestras clases dibujadas de este modo.



En la imagen superior vemos el modo en que dibujaríamos las clases de nuestros problema. El campo para los métodos de momento lo he dejado vacío. En los atributos, antes de su nombre, he puesto el símbolo - para indicar que se trata de un atributo privado y cumplir así los criterios de

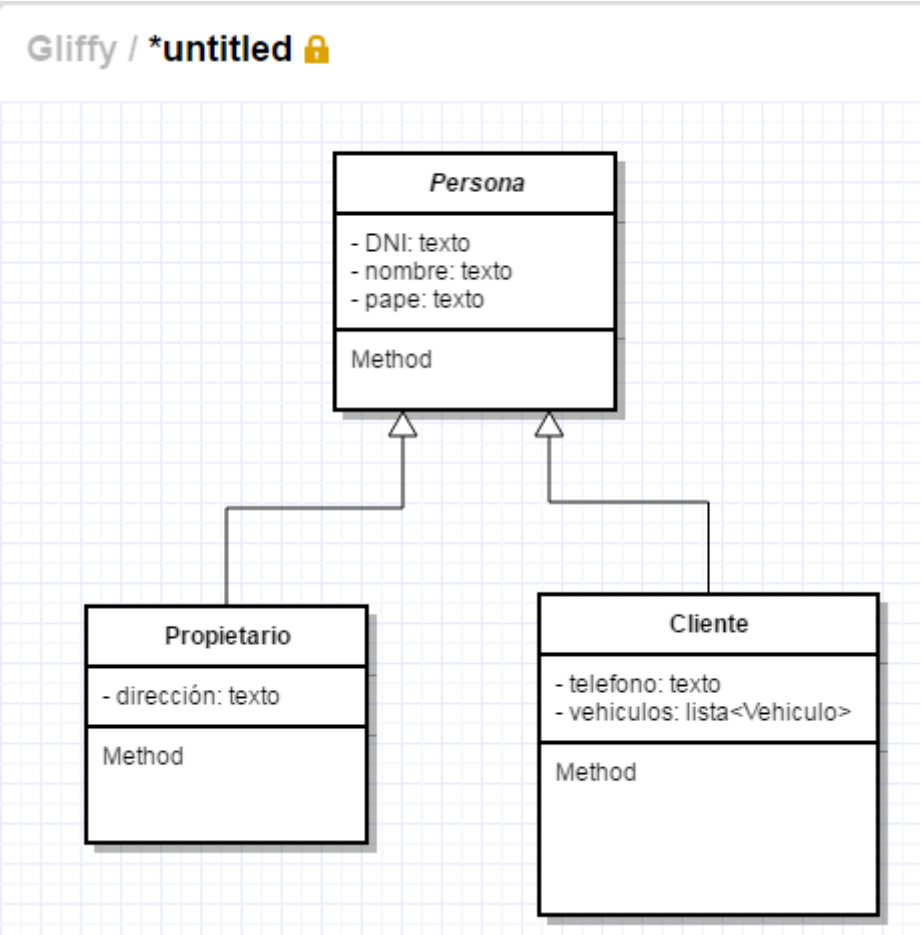
encapsulación cuando cree los getters y setters correspondientes.

Podemos crear diagramas de clases con la aplicación web https://www.gliffy.com/uses/uml-software/

En el siguiente apartado vamos a redefinir este diagrama para ver si de algún modo existe herencia y las distintas relaciones entre las clases.

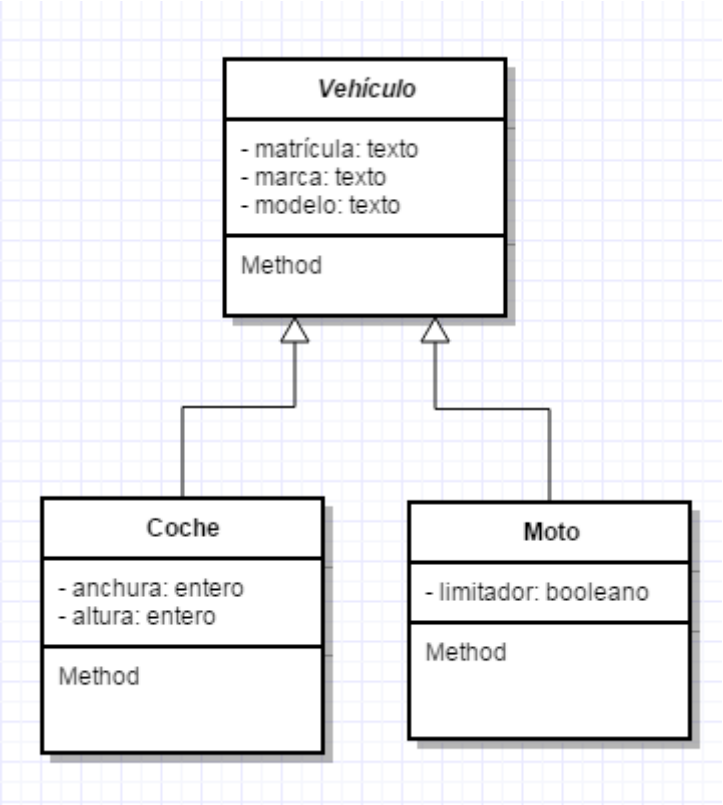
Jerarquía de clases

Si analizamos el diagrama de clases que hemos realizado en el apartado anterior observamos que los clientes y los propietarios comparten parte de sus atributos por lo que podríamos crear una abstracción que llamásemos Persona que tuviese estos campos y que posteriormente Propietario y Cliente heredasen de ella. Vamos a ver como quedaría.



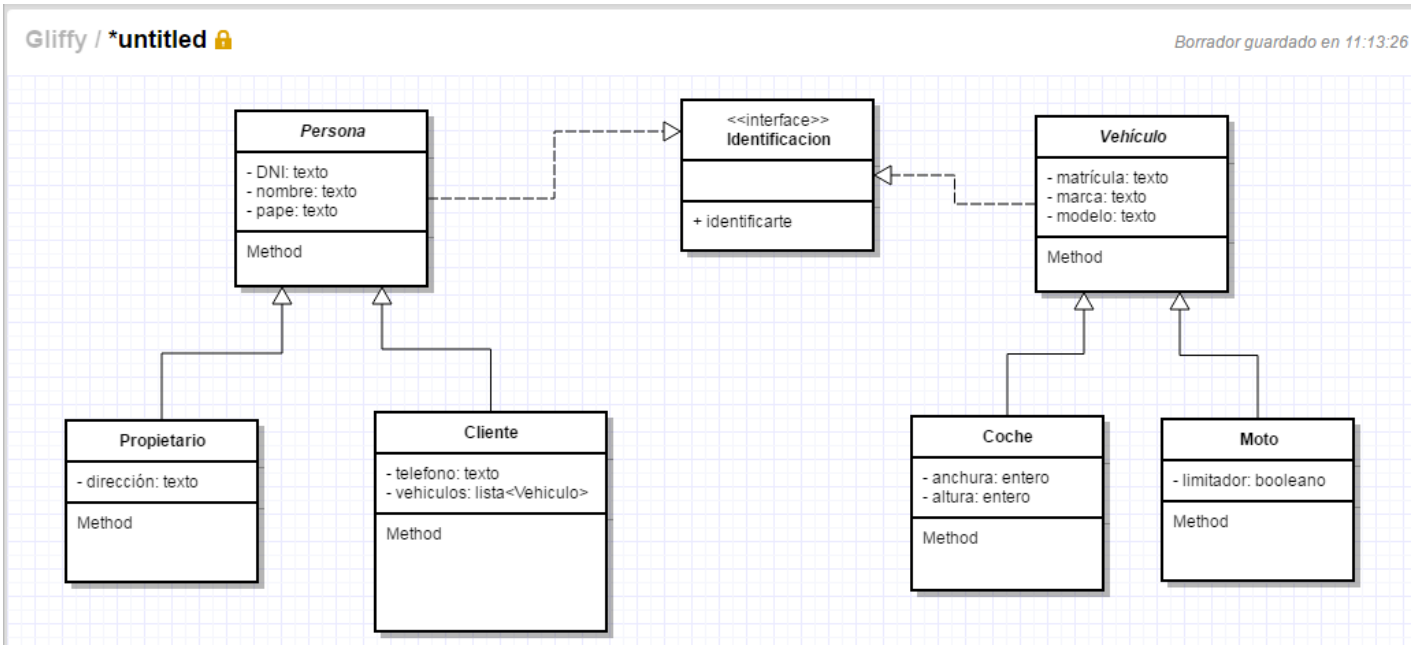
Fíjate que en la imagen anterior la clase Persona, al considerarla abstracta, escribimos su nombre con letra cursiva. Consideramos la clase Persona abstracta porque no vamos a tener que crear en ningún momento instancias de de ella.

De modo análogo ocurre con Vehículo, Coche y Moto. A fin de cuentas comparten una serie de atributos, podemos "sacar factor común" haciendo que Coche y Moto hereden de Vehículo tal y como vemos en la imagen posterior.



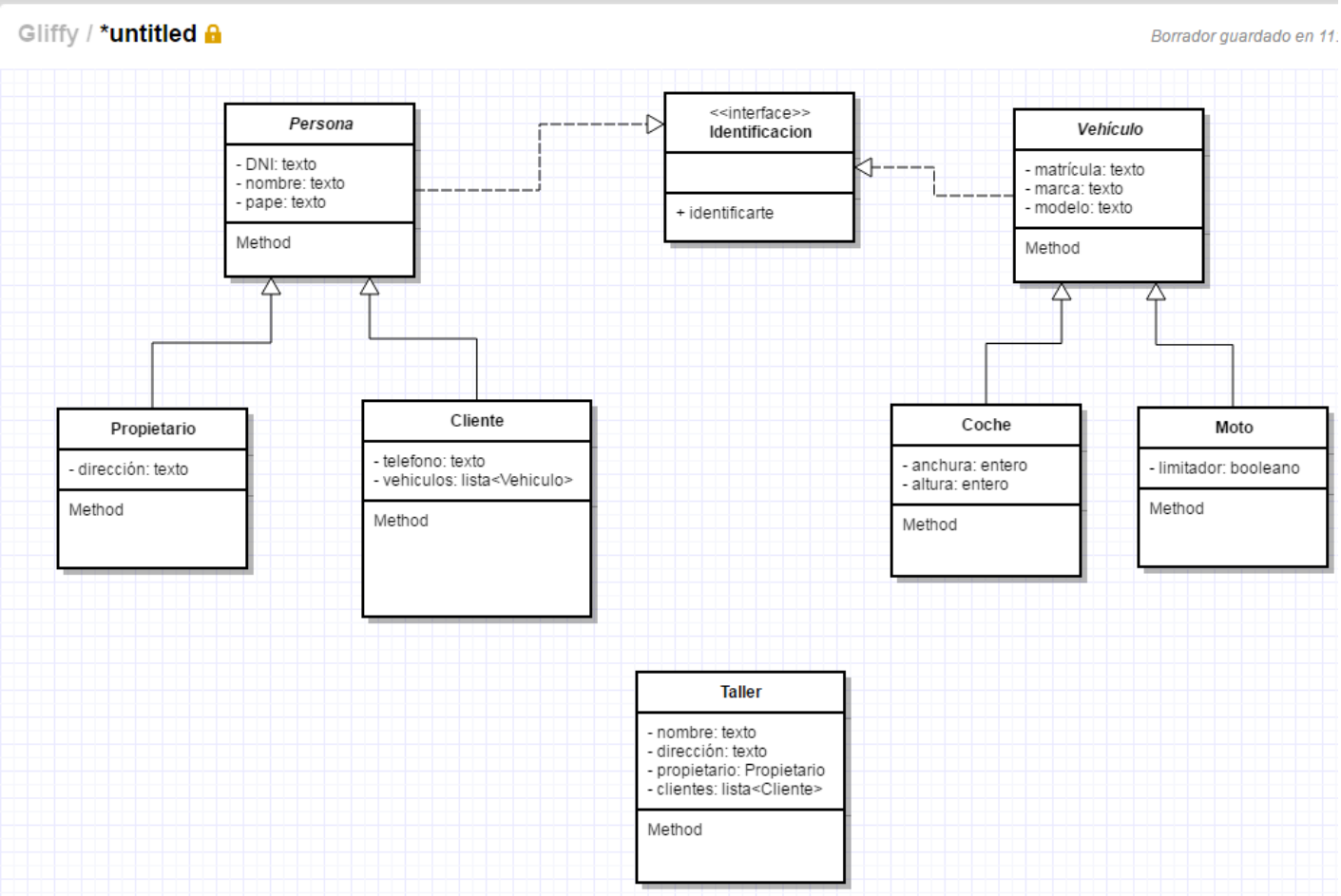
Igual que antes Vehículo es una clase abstracta y como tal escribimos su nombre en cursiva. Coche y Moto extenderán a Vehículo.

En el enunciado de nuestro problema se dice "Los propietarios, clientes y vehículos deberán implementar una funcionalidad que los haga identificarse.", esto debería ser un método pero ¿en que clase lo coloco? Es algo que tanto las personas como los vehículos deben implentar ¿pongo el método una vez en cada clase? La solución a nuestras dudas es crear una interface que vamos a llamar Identicación y haremos que tanto Persona como Vehículo implementen esta clase. Vamos a ver como dibujaríamos esta interface y la relación entre ella, Persona y Vehículo.

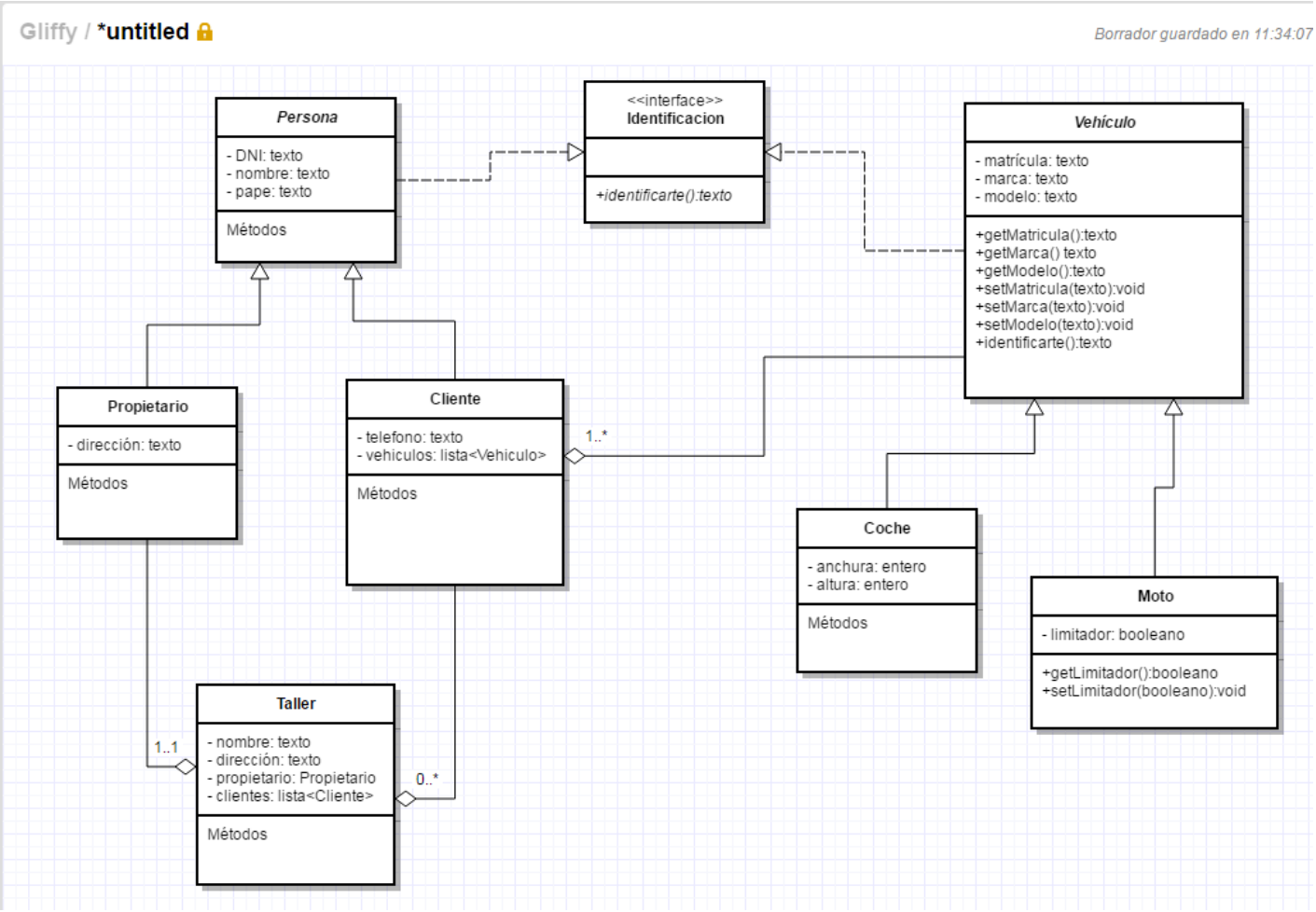


Observa que la línea entre las clases es en este caso discontinua mientras que cuando expresamos herencia es una línea continua.

Si comparamos nuestro esquema actual con nuestro primer esquema observaremos que nos falta la clase Taller así que vamos a añadirla a nuestro esquema.



Casi hemos terminado. De algún modo tenemos que expresar la composición entre clases y la cardinalidad existente. Es decir, si nos fijamos en la clase Cliente vemos que está compuesta por una lista de Vehículos, tenemos que expresarlo de algún modo, además, cada cliente deberá tener al menos 1 vehículo. Si nos fijamos en Taller vemos que está compuesto tanto por Propietario (cada taller tiene 1 único propietario) como por una lista de clientes que serán 0 o mas. Vamos a ver como representar esta composición y cardinalidad.



Con esto tendríamos terminado nuestro esquema de clases donde podemos observar la jerarquía de clases existente y las relaciones que existen entre ellas. Faltaría añadir en cada clase los métodos que tiene donde pone Métodos. Únicamente los he añadido en las clases Identificacion, Vehículo y Moto.

Con esto únicamente nos quedaría codificar nuestro esquema, pero esto lo haremos en el siguiente capítulo.

Codificación

Una vez que tenemos esquematizada la solución de nuestro programa es el momento de codificarla. En esta sección voy a añadir imágenes de las partes que considero mas relevantes para solucionar nuestro problema. En el apartado "Código utilizado en los ejemplos" podréis descargar el código completo del proyecto.

```
1  package catedusc.taller.interfaces;
2
3  /**
4   * @author Pablo Ruiz Soria
5   */
6  public interface Identificacion {
7      public String identificarte();
8  }
```

El código anterior es la implementación de la interface Identificación. En ella vemos que aparece la firma de la función identificarte. Como ya sabemos a estas alturas todas las clases que implementen esta interface estarán obligadas a implementar esta función.

Vamos a contnuar con la clase abstracta Persona que implementa la interface anterior.

```
1  package catedusc.taller.humano;
2
3  import catedusc.taller.interfaces.Identificacion;
4  import catedusc.taller.util.Comprobador;
5
6  /**
7   * @author Pablo Ruiz Soria
8   */
9  public abstract class Persona implements Identificacion {
10     private String dni;
11     private String nombre;
12     private String pape;
13
14     public Persona(String dni, String nombre, String pape) throws Exception{
15         if(Comprobador.esValidoDNI(dni) ){
16             this.dni = dni;
17         }else{
18             throw new Exception("DNI inválido");
19         }
20         this.nombre = nombre;
21         this.pape = pape;
22     }
23
```

Como vemos en esta clase indicamos que es una clase pública y abstracta que implementa la interface Identificacion. En esta clase tenemos 3 atributos que siguen los principios de la encapsulación y en la imagen vemos que la clase tiene un único constructor que obliga a definir una Persona a través de 3 argumentos. Como ya sabemos, al tratarse de una clase abstracta no podremos crear objetos de esta clase. La clase Vehículo sería similar a esta clase que acabamos de analizar.

Vamos a continuar con el código de la clase Propietario que hereda de la clase anterior y se verá obligada a implementar el método de la interface Identificacion.

```
1   package catedusc.taller.humano;
2
3   /**
4    * @author Pablo Ruiz Soria
5    */
6   public class Propietario extends Persona {
7       private String direccion;
8
9       public Propietario(String dni, String nombre, String pape, String direccion)
10          super(dni, nombre, pape);
11          this.direccion = direccion;
12      }
13
14      /**
15       * @return the direccion
16       */
17      public String getDireccion() {
18          return direccion;
19      }
20
21      /**
22       * @param direccion the direccion to set
23       */
24      public void setDireccion(String direccion) {
25          this.direccion = direccion;
26      }
27
28      @Override
29      public String identificarte() {
30          return "Soy un propietario. " + super.identificarte() + " y mi dirección
31                  + this.getDireccion();
32      }
33  }
```

Vemos en la línea 6 que se trata de una clase pública que extiende a la clase Persona (que a su vez implementaba a Identificacion). Al heredar de Persona dispondremos de acceso a aquellos atributos y métodos no privados. Si nos fijamos en el constructor vemos que hemos de llamar a super (línea 10) para que se llame al constructor de la superclase de Propietario (Persona). Observamos también en las líneas 28 a 32 la implementación del método de la interface. Del mismo modo que construimos la clase Propietario procederíamos con la clase Cliente.

Una vez hemos creado todas nuestras clases únicamente necesitaríamos disponer de una clase con un método main que nos permitiese interactuar con las clases que hemos definido para poder crear objetos a partir de ellas. En el apartado "Código utilizado en los ejemplos" he añadido 2 clases que nos permiten lanzar la aplicación, la clase Inicio.java contiene una interfaz de tipo texto y la clase igu contiene una interfaz gráfica de usuario limitada. El código se encuentra documentado para facilitar su comprensión.

En el siguiente apartado vamos a ver como crear esta interfaz gráfica de usuario a la que hago referencia en el párrafo anterior.

Interfaz gráfica de usuario

Hasta el momento siempre que hemos introducido algún dato lo hemos hecho a través del código de nuestros programas pero lo habitual es que los programas tengan una interfaz gráfica de usuario (IGU, GUI en inglés) con la que nosotros interactuemos y de ese modo introduzcamos u obtengamos datos. Por ello he preparado el siguiente videotutorial en el que muestro como crear una IGU básica por si queremos incorporarla a nuestros programas.

https://www.youtube.com/embed/jQwbk5A3Lxc?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq

Calculadora con interfaz gráfica de usuario y ejecutable

En este apartado vamos a ver como elaborar una calculadora que tenga una interfaz gráfica de usuario y como generar un "fichero ejecutable" que nos permita ejecutar nuestros proyectos en cualquier sistema operativo que disponga de la JRE. Os dejo el vídeo a continuación:

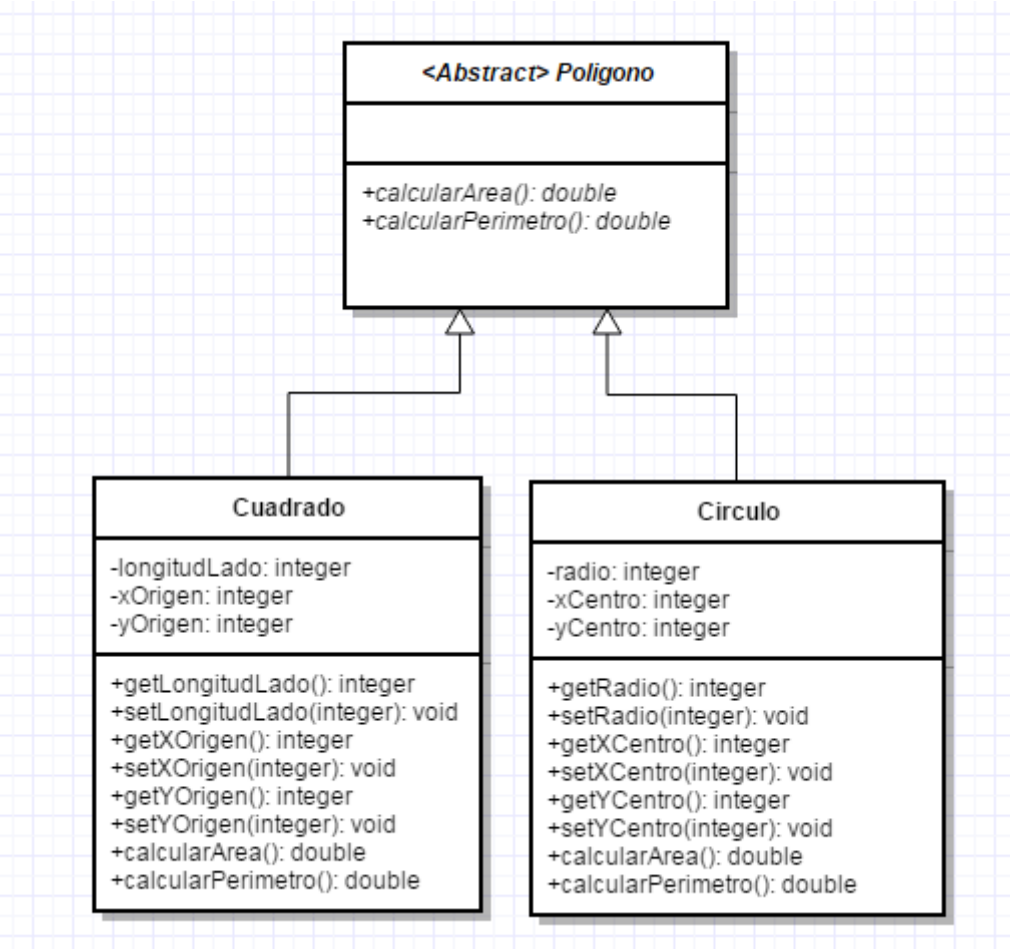
https://www.youtube.com/embed/9xKSFonCYB8?list=PL1ubUMkNBAR_98ka0Q393l3fpzSjo3FGq

Pintemos en Java

En este vídeo vamos a trabajar con las características de la POO (clases abstractas, herencia, polimorfimo) y vamos a hacerlo a través de una interfaz gráfica de usuario. Además, en esta interfaz gráfica de usuario, vamos a dibujar los objetos con los que vamos a trabajar. El enunciado de nuestro problema dice:

> Para trabajar la unidad didáctica de geometría vamos a desarrollar una aplicación informática que nos permita trabajar con polígonos, concretamente queremos trabajar con cuadrados y cículos. De cualquier polígono queremos poder calcular su área y perímetro. De los cuadrados queremos almacenar su longitud de lado y el punto (x e y) que lo define en su posición superior izquierda. De los círculos queremos almacenar su centro (x e y) y su radio. En la aplicación deberemos de poder crear cualquiera de estos 2 elementos, listarlos (viendo el área de los cuadrados y el perímetro de los círculos) y dibujar todos los polígonos que tengamos. Todo hecho deberemos hacerlo haciendo un buen uso de las características de la POO. Por simplicidad, no es necesario controlar las excepciones ni que los valores de los lados son positivos.

El diagrama de clases resultante del enunciado anterior sería el siguiente:



Vamos a ver como implementarlo en el siguiente vídeo:

https://www.youtube.com/embed/o330-623xrI

Bibliografía recomendada

Soy consciente de que quizás el curso no haya resuelto todas tus dudas acerca de la programación orientada a objetos y el lenguaje de programación Java puesto que se pretende que el mismo sea una introducción, por ello no querría dejar pasar la ocasión de recomendarte un par de libros que creo pueden resultarte de utilidad además de la gran cantidad de información existente en internet.

El primero de los libros se titula "Piensa en Java" escrito por Bruce Eckel. Aunque la última edición data de 2006 sigue siendo un libro de referencia para quienes quieren iniciarse en la programación orientada a objetos y el lenguaje de programación Java. En su página oficial (inglés) podemos encontrar mas información sobre el mismo.

El segundo libro es un libro de programación avanzada orientado a la ingeniería del software. El libro es interesante ya que nos proporciona una buena visión sobre buenas prácticas y nos plantea reflexiones que nos harán mejorar en el modo en que escribimos nuestro código haciéndonos mas eficientes. Su título es "Código límpio" y está escrito por Robert C. Martin. Esta es su página web para la edición en castellano.

Código utilizado en los ejemplos

Módulo 4 Taller (zip - 0.04 MB).

Terminando

Tarea

La cooperativa de ebanistas de CATEDU nos ha indicado que necesita una aplicación informática para la gestión de su ebanistería. En esta ebanistería fabrican muebles. De los muebles quieren guardarse sus dimensiones (ancho, alto y profundo) en centímetros como números enteros. Los tipos de muebles que fabrican son mesas de oficina, mesas de taller y estanterías. De todas las mesas se quiere conocer si tienen cajonera o no pero de las mesas de oficina se quiere conocer el material con el que están hechas mientras que de las mesas de taller se quiere almacenar su grado de resistencia (un número del 1 al 10). De las estanterías se quiere guardar el número de baldas.

A partir del enunciado anterior tu tarea consiste en:

- Crear un diagrama de clases en el que se reflejen las clases de nuestro problema así como los atributos de las mismas y las relaciones de herencia y composición existentes en caso de haberlas
- Implementar con Java el diagrama que has realizado

Terminando

Voluntario: Muestra tus proyectos

https://padlet.com/embed/fljmvkdbwmh6

Hecho con Padlet

Image not found or type unknov

Créditos

Autoría

- {{ book.author }}

Cualquier observación o detección de error en soporte.catedu.es