DESARROLLO DE APLICACIONES MULTIPLATAFORMA

PROGRAMACIÓN

TEMA 6

DESARROLLO DE CLASES

CONTENIDOS

6.1. CREACIÓN DE PAQUETES.

Un paquete es un conjunto de clases relacionadas entre sí, ordenadas de forma arbitraria teniendo en cuenta que:

- Un paquete puede contener a su vez “subpaquetes”.
- Java mantiene su biblioteca de clases en una estructura jerárquica.
- Cuando nos referimos a una clase de un paquete (salvo que se haya importado el paquete) hay que referirse a la misma especificando el nombre del paquete (y subpaquete si es necesario) al que pertenece.
- Los paquetes permiten reducir los conflictos con los nombres puesto que dos clases que se llaman igual, si pertenecen a paquetes distintos, no deberían dar problemas.
- Los paquetes permiten proteger ciertas clases no públicas del acceso desde fuera del mismo.
- Favorecen la reutilización y el mantenimiento del código.
- La estructura/jerarquía de paquetes dentro de un proyecto está directamente relacionada con la estructura de carpetas/directorios del proyecto.

6.1. CREACIÓN DE PAQUETES.

En los ficheros de código Java se usa la palabra reservada “package” para especificar a qué paquete pertenecen. Suele indicarse como primera sentencia.

La sentencia “import” nos permite utilizar una o varias clases contenidas en un paquete dentro de otra clase que se encuentra fuera del paquete.

Para encontrar una clase u otro recurso Java necesitamos dos cosas:

- *El nombre del paquete.*
- *Las rutas donde están situados los paquetes y las clases.*

Al igual que la variable de entorno PATH nos permite definir dónde está situado el directorio “bin” del JDK para poder compilar y ejecutar nuestro programa, la variable de entorno CLASSPATH nos permite definir dónde están situadas las clases externas a las de la API que necesitemos para compilar y ejecutar nuestro programa.

6.1. CREACIÓN DE PAQUETES.

Establecer el valor de esta variable es necesario cuando se va a utilizar una clase que no está en el mismo directorio (o subdirectorios) que la clase con la que se está trabajando, o no está en ningún lugar definido por el mecanismo de extensiones de Java (*jre/lib/ext*).

Una alternativa a establecer el valor de la variable de entorno CLASSPATH es utilizar las opciones *-cp* o *-classpath* al ejecutar Java (*java, javac, javah o jdb*).

Ej.:

set CLASSPATH="C:\Proyecto"
javac miproyecto.java
java miproyecto

javac -cp "C:\Proyecto" miproyecto.java
java miproyecto

6.1. CREACIÓN DE PAQUETES.

Respecto a los niveles de acceso en Java ya vistos:

public, protected, private y "no especificado" .

Hay que decir, por su relación con los paquetes, que los miembros de una clase (atributos o métodos) con nivel "no especificado" podrán ser accedidos por cualquier clase perteneciente al mismo paquete.

Es decir, un miembro de una clase "no etiquetado" se comporta como público para las clases del mismo paquete y como privado para las clases de fuera del paquete.

Además, una clase definida como pública puede ser utilizada por las clases de su paquete y otros paquetes, mientras que una clase "no etiquetada" solamente podrá ser utilizada por las clases de su propio paquete.

6.1. CREACIÓN DE PAQUETES.

Las aplicaciones profesionales suelen contener múltiples ficheros de datos y para distribuirlas se suele utilizar un fichero JAR. Este formato permite empaquetar múltiples ficheros (clases, datos, sonido, imágenes, etc.) en un solo archivo y tiene muchas ventajas.

Dado que están comprimidos, se pueden descargar las aplicaciones de una sola vez, proporciona mecanismos de seguridad y ofrece una gran portabilidad, al ser un estándar muy común de la plataforma Java.

- Crear un fichero JAR:
  jar cf ejemplo.jar ejemplo.class
- Ver el contenido del JAR:
  jar tf ejemplo.jar
- Extraer los ficheros del JAR:
  jar xf ejemplo.jar
- Ejecutar la aplicación contenida en un JAR:
  java -cp ejemplo.jar ejemplo

6.2. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

En Java no existen “variables globales” (como en C o C++), por lo tanto, si queremos utilizar una variable única que puedan utilizar todos los objetos de una clase deberemos declararla como estática (*static*).

A los atributos estáticos de una clase se les llama atributos de clase, y a los atributos no estáticos o “normales”, se les llama atributos de instancia.

De igual manera, los métodos de una clase pueden ser:

- Métodos de instancia: Son aquellos utilizados por la instancia. Cada instancia u objeto tendrá sus propios métodos independientes del mismo método de otro objeto de la misma clase.
- Métodos de clase: Son aquellos comunes para una clase (un método por clase, no por objeto).
  - Los métodos *static* no tienen referencia *this*.
  - Un método *static* no puede acceder a miembros que no sean *static*.
  - Un método no *static* sí puede acceder a miembros *static* y no *static*.

6.2. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

Una clase, método o atributo declarado como estático puede ser accedido o invocado sin la necesidad de tener que instanciar un objeto de la clase. Los atributos declarados como estáticos son inicializados en el momento en que se carga la clase en memoria.

En conjunción con la directiva “final”, que evita la modificación del atributo (constante), las constantes de clase son definidas de esta manera: *static final*.

Es posible declarar bloques de código como estáticos, de tal manera que sean ejecutados cuando se cargue la clase. Este tipo de bloques se conocen como bloques de inicialización estáticos. Si no se declara un bloque de este tipo de forma explícita, el compilador combina todos los atributos estáticos en un bloque y los ejecuta durante la carga de la clase.

```
public class Test {
    private static int atributo1;
    static {
        atributo1 = 10;       // En principio, inicializar aquí o directamente sería equivalente,
    }     // pero dentro de un bloque podemos hacer otras operaciones.
}
```

6.2. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

```
class Alumno {

    private String nombre;                  // Miembro de instancia
    private static int numAlumnos=0;        // Miembro de clase

    public Alumno() {
        nombre="";
        numAlumnos++;
    }

    public String getNombre() { return nombre; }
    public int getNumAlumnos() { return numAlumnos; }
    public void setNombre(String nombre) { this.nombre=nombre; }
    // NO TIENE SENTIDO HACER UN MÉTODO SET DE UN STATIC
    // public void setNumAlumnos(int numAlumnos) { this.numAlumnos=numAlumnos; }

    public static void metodoEstatico() {
        //nombre="";                    // Fallo de compilación
        //this.numAlumnos=0;            // Fallo de compilación
        numAlumnos=0;
        //String s=getNombre(); // Fallo de compilación
    }
}
```

6.2. ESTRUCTURA Y MIEMBROS DE UNA CLASE.

```
public class Test03_static {

    public static void main(String[] args) {

        System.out.println("");

        Alumno a1=new Alumno();
        Alumno a2=new Alumno();
        Alumno a3=new Alumno();

        System.out.println(a1.getNumAlumnos());     // Muestra un 3
        System.out.println(a3.getNumAlumnos());     // Muestra un 3

        //Alumno.numAlumnos=0;       // En este caso falla porque es privado.
        Alumno.metodoEstatico();     // Uso correcto de un método estático,
                                     // Anteponiendo el nombre de la clase.

        System.out.println("");
    }
}
```

6.3. TRABAJANDO CON MÉTODOS.
6.3.1. PASO DE PARÁMETROS.

En ocasiones, necesitamos que las variables que pasamos como parámetros cambien su valor una vez ejecutado el método, si éste las ha modificado antes.

Si es una única variable se puede solucionar con la sentencia “return”, pero si pasamos varias variables y queremos conservar los valores modificados por el método, entonces con las sentencia “return” no es suficiente.

Existen dos tipos de paso de parámetros:

- Paso de parámetros por valor. Las variables se copian en los parámetros del método. Las variables pasadas como parámetro no se modifican.
- Paso de parámetros por referencia. Las variables pasadas como parámetros se modifican puesto que el método trabaja con las direcciones de memoria de los parámetros (referencias o punteros).

Java utiliza siempre paso por valor (o copia), pero en el caso de parámetros que son objetos (todos menos los datos primitivos), lo que se copia es una referencia, por eso funciona como si fuera paso por referencia.

6.3. TRABAJANDO CON MÉTODOS.

6.3.1. PASO DE PARÁMETROS.

```
class Alumno {

    private String nombre;

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

public class Test04_parametros {

    public static int incrementar1(int n) {
        return ++n;
    }

    public static int incrementar2(int[] n) {
        return ++n[0];
    }

    public static String cambiarNombre1(Alumno b) {
        b.setNombre("manolo");
        return b.getNombre();
    }

    public static String cambiarNombre2(Alumno b) {
        Alumno c=new Alumno();
        b=c;
        b.setNombre("manolo");
        return b.getNombre();
    }
```

6.3. TRABAJANDO CON MÉTODOS.

6.3.1. PASO DE PARÁMETROS.

```
public static void main(String[] args) {

    System.out.println("");

    int x=1;

    System.out.println(incrementar1(x));    // Paso de parámetro por valor
    System.out.println(x);

    System.out.println("");

    int[] y={1};                            // Equivale a int[] y=new int[1]; y[0]=1;

    System.out.println(incrementar2(y));    // Paso de parámetros por referencia
    System.out.println(y[0]);               // ¿ Qué pasa si hago System.out.println(y); ?


    // ¿ Qué sucede si en vez de tipos primitivos utilizamos objetos ?

    System.out.println("");

    Alumno a=new Alumno();
    a.setNombre("pepe");
    System.out.println(cambiarNombre1(a));
    System.out.println(a.getNombre());

    System.out.println("");

    Alumno aa=new Alumno();
    aa.setNombre("pepe");
    System.out.println(cambiarNombre2(aa));
    System.out.println(aa.getNombre());
```

6.3. TRABAJANDO CON MÉTODOS.
6.3.1. PASO DE PARÁMETROS.

Casos especiales sin paso por referencia: Clases inmutables vs mutables.

- La clase *String* y los *Wrappers* son inmutables, es decir, no cambian su valor una vez creados (entre otras razones pq no disponen de métodos *Setters*).
- La clase *String* tiene alternativas mutables, como son las clases *StringBuffer* y *StringBuilder*.

```
public static String cambiar1(String r) {
     r="manolo";
     return r;
}

public static Integer cambiar2(Integer o) {
     o=2;
     return o;
}

public static void main(String[] args) {

     System.out.println("");

     String s="pepe";                    // Similar a String s=new String("pepe");
     System.out.println(cambiar1(s));
     System.out.println(s);

     System.out.println("");

     Integer i=1;
     System.out.println(cambiar2(i));
     System.out.println(i);
```

6.3. TRABAJANDO CON MÉTODOS.

6.3.1. PASO DE PARÁMETROS.

Podemos transformar una clase cualquiera en inmutable, simplemente, eliminando sus métodos *Setters*.

```
    public static String cambiarNombre3(AlumnoInMutable b) {
        //b.setNombre("manolo");     // No existe Set,
        return b.getNombre();        // esta es la razón por la que es inmutable!!
    }

    public static void main(String[] args) {

        System.out.println("");

        AlumnoInMutable aim=new AlumnoInMutable("pepe");
        System.out.println(cambiarNombre3(aim));
        System.out.println(aim.getNombre());

        System.out.println("");
    }

}

class AlumnoInMutable {

    // No tiene métodos Set, la asignación se hace a través del constructor.
    private String nombre;

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

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

6.3. TRABAJANDO CON MÉTODOS.
6.3.2. SOBRECARGA DE MÉTODOS.

La sobrecarga de un método es la implementación del método varias veces con ligeras diferencias adaptadas a las distintas necesidades de dicho método.

Para crear métodos sobrecargados debemos crear métodos con el mismo nombre pero con distinta lista de parámetros y se deben cumplir las siguientes reglas:

- *Los métodos sobrecargados deben de cambiar la lista de parámetros obligatoriamente (la cantidad, el tipo o el orden).*
- *Los métodos sobrecargados pueden cambiar el tipo de retorno o el modificador de acceso (pero no únicamente).*
- *Un método puede estar sobrecargado en la clase o en una subclase.*
- *Al sobrecargar un método se pueden utilizar las mismas excepciones o añadir nuevas.*

6.3. TRABAJANDO CON MÉTODOS.

6.3.2. SOBRECARGA DE MÉTODOS.

```
/*public*/ class Alumno {

    private String nombre;
    private String apellido1;
    private String apellido2;
    private double edad;

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

    public void setNombre(String nombre, String apellido1) {
        this.nombre=nombre;
        this.apellido1=apellido1;
    }

    public void setNombre(String nombre, String apellido1, String apellido2) {
        this.nombre=nombre;
        this.apellido1=apellido1;
        this.apellido2=apellido2;
    }

    public String getNombre() { return nombre+" "+apellido1+" "+apellido2; }
```

6.3. TRABAJANDO CON MÉTODOS.

6.3.2. SOBRECARGA DE MÉTODOS.

```
    public void setEdad(int edad) {
        this.edad=edad; //cast implícito
    }

    public void setEdad(double edad) {
        this.edad=edad;
    }

    public double getEdad() { return edad; }

    //public int getEdad() { return (int)edad; }    //Provoca error de compilación!!
}

public class Test05_sobrecargametodos {

    public static void main(String[] args) {

        System.out.println("");

        Alumno a=new Alumno();

        a.setNombre("pepe");
        a.setNombre("pepe","martinez");
        a.setNombre("pepe","martinez","sanchez");
        System.out.println(a.getNombre());

        a.setEdad(20);
        a.setEdad(20.5);
        System.out.println(a.getEdad());

        System.out.println("");
    }
}
```

6.3. TRABAJANDO CON MÉTODOS.
6.3.3. LISTA DE ARGUM. DE LONG. VARIABLE.

Las listas de argumentos de longitud variable permiten crear métodos que reciben un número arbitrario de argumentos.

Un tipo de argumento precedido de una "elipsis" (…) indica que el método recibe un número variable de argumentos de ese tipo. El uso de la "elipsis" solo puede ocurrir una vez en la lista de parámetros. La "elipsis" con su tipo debe colocarse como último parámetro del método.

Java trata a la lista de argumentos de longitud variable como un array.

Parámetros vs Argumentos

- *Un parámetro representa un valor que el método espera que se transfiera cuando es llamado. La declaración del método define sus parámetros. El nombre de cada parámetro actúa como una variable local en el método.*
- *Un argumento representa el valor que se transfiere a un parámetro del método cuando se llama al método. El código de llamada proporciona los argumentos cuando llama al método. Los argumentos no tienen nombre y se corresponden con el parámetro situado en la misma posición de la lista.*

6.3. TRABAJANDO CON MÉTODOS.

6.3.3. LISTA DE ARGUM. DE LONG. VARIABLE.

```
public class Test06_listaargumentoslongvariable {

    public static void main(String[] args) {

        System.out.println("");

        System.out.printf("%.2f\n",promedio(1));
        System.out.printf("%.2f\n",promedio(1,2,5));
        System.out.printf("%.2f\n",promedio(1,2,3,4,7));

        System.out.println("");
    }

    private static double promedio(double... numeros) {
        double total=0.0;
        for (int i=0; i<numeros.length; i++)
            total=total+numeros[i];
        return total/numeros.length;
    }

}
```

6.3. TRABAJANDO CON MÉTODOS.

6.3.4. CONSTRUCTORES Y DESTRUCTORES.

Java, al igual que hace con las variables, cuando va a crear un objeto, lo que hace es reservar memoria para dicho objeto. En esta fase de construcción del objeto, Java crea un constructor público por defecto, no obstante, podemos crear más constructores para la misma clase.

Por lo tanto, hay dos tipos de constructores:

- Constructor por defecto. Cuando no se especifica en el código. Se ejecuta siempre de manera automática e inicializa el objeto con los valores predeterminados del sistema.
- Constructor definido. Puede ser más de uno. Tiene el mismo nombre de la clase. Nunca devuelve un valor y no puede ser declarado como *static, final, native, abstract o synchronized*. Inicializa el objeto con los valores especificados o si no, con los predeterminados del sistema. Por regla general, se declaran los constructores como públicos para que puedan ser accedidos por otra clase.

Cuando existe más de un constructor para una clase se dice que éste está sobrecargado. Cuando creamos un objeto con "new", Java elige el constructor más adecuado dependiendo de los parámetros utilizados.

6.3. TRABAJANDO CON MÉTODOS.

6.3.4. CONSTRUCTORES Y DESTRUCTORES.

Cuando trabajamos con objetos estamos trabajando con referencias. Una referencia es una localización o posición en memoria donde se encuentra el objeto.

Cuando se asigna (=) un objeto a otro del mismo tipo, lo que se asigna es la referencia de un objeto al otro, no se copian los valores de sus atributos, por lo que tendremos dos variables que referencian (apuntan) al mismo objeto.

Si lo que queremos es copiar los valores de los atributos de un objeto en otro objeto del mismo tipo, podemos asignarlos uno a uno o utilizar un constructor de copia.

Con un constructor de copia se inicializa un objeto asignándole los valores de otro objeto diferente de la misma clase. Este constructor de copia tendrá solo un parámetro, un objeto de la misma clase.

6.3. TRABAJANDO CON MÉTODOS.

6.3.4. CONSTRUCTORES Y DESTRUCTORES.

```
/*public*/ class Alumno {

    private String nombre;

    public Alumno() {                   // Constructor 1
        nombre="";
    }

    public Alumno(String nombre) {  // Constructor 2 (existe sobrecarga)
        this.nombre=nombre;
    }

    public Alumno(Alumno a) {           // Constructor de copia
        nombre=a.getNombre();
    }

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

6.3. TRABAJANDO CON MÉTODOS.

6.3.4. CONSTRUCTORES Y DESTRUCTORES.

```
public class Test07_constructor {

    public static void main(String[] args) {

        System.out.println("");

        Alumno a1=new Alumno();              // Constructor básico
        Alumno a2=new Alumno("pepe");        // Constructor sobrecargado

        a1=a2;
        a1.setNombre("manolo");
        System.out.println(a2.getNombre()); // ¿ Qué nombre muestra por pantalla ?

        Alumno a3=new Alumno("pepe");        // Constructor sobrecargado
        Alumno a4=new Alumno(a3);            // Constructor de copia
        a3.setNombre("manolo");

        System.out.println(a3.getNombre());
        System.out.println(a4.getNombre()); // ¿ Qué nombres muestra por pantalla ?

        System.out.println("");
    }
}
```

6.3. TRABAJANDO CON MÉTODOS.
6.3.4. CONSTRUCTORES Y DESTRUCTORES.

Los destructores de una clase no existen en Java.

Al contrario que en otros lenguajes OO como C++, es el propio sistema el que se encarga de eliminar definitivamente los objetos de memoria cuando le asignamos el valor “null” a la referencia, le asignamos un objeto diferente o bien, termina el bloque de código donde está definida la referencia (ámbito).

Cuando se va a liberar automáticamente la memoria de objetos inservibles, el sistema ejecuta el finalizador de los objetos:

protected void finalize();

Podemos sobreescribir este método para realizar otras acciones adicionales (liberar memoria, cerrar ficheros, conexiones, etc.). Pero no se sabe cuando se va a ejecutar por lo que las operaciones para liberar otros recursos mejor realizarlas de forma explícita en un método.

6.3. TRABAJANDO CON MÉTODOS.

6.3.4. CONSTRUCTORES Y DESTRUCTORES.

El sistema de liberación de memoria en Java se llama *garbage collector* (recolector de basura) que trabaja de forma automática.

Se ejecuta en segundo plano en un subproceso paralelo a la propia aplicación.

Es posible sugerir que se active realizando la llamada a los siguientes métodos:

System.runFinalization();

System.gc();

6.3. TRABAJANDO CON MÉTODOS.
6.3.5. RECURSIVIDAD.

Un método se llama recursivo cuando se llama a sí mismo.

¿Cuándo utilizar la recursividad?

- Cuando la resolución del problema es más sencilla que su versión iterativa.
- Cuando no es infinita, es decir, hay un caso resoluble más básico o más sencillo.
- Cuando se va a resolver un problema recursivo, en cada llamada sucesiva al método recursivo nos debe ir acercando a la solución.

¿Es eficiente la recursividad?

- NO. Siempre hay un método equivalente iterativo (aunque sea más complicado de programar y entender).

6.3. TRABAJANDO CON MÉTODOS.

6.3.5. RECURSIVIDAD.

```
//  2³ = 2*2*2 = 8
//  2³ = 2*2² = 2*2*2¹ = 2*2*2 = 8

public static int potenciaR(int x, int y) {

    if (y==1) {                          // Caso base
        return x;
    } else {                             // Reducción de la complejidad
        return x*potenciaR(x,y-1);
    }
}

public static int potenciaI(int x, int y) {

    //~ int resultado=1;                              // Implementación 1
    //~ for (int j=0; j<y; j++)
        //~ resultado=resultado*x;
    //~ return resultado;

    int resultado=1;                                  // Implementación 2
    while (y>0) {
        resultado=resultado*x;
        y--;
    }
    return resultado;
}
```

6.4. WRAPPERS.

La principal diferencia entre un tipo primitivo y un *wrapper* (envoltorio) es que este último es una clase. Cuando trabajamos con *wrappers* estamos trabajando con objetos mientras que cuando trabajamos con un tipo primitivo obviamente no.

A simple vista un *wrapper* siendo un objeto, puede aportar muchas ventajas pero el problema está cuando le pasamos una variable a un método como parámetro, si esta variable es de un tipo primitivo se le pasa por valor mientras que si es un *wrapper* se le pasa como "referencia", mejor dicho, copia la referencia. Pero en el caso de los *wrapper* no se actualiza el valor ya que no tenemos métodos que hagan tal cosa dentro de la clase del *wrapper*.

Existen *wrapper* de todos los tipos primitivos:

byte	*Byte*	*float*	*Float*
short	*Short*	*double*	*Double*
int	*Integer*	*char*	*Character*
long	*Long*	*boolean*	*Boolean*

6.4. WRAPPERS.

Ejemplo: Métodos del wrapper Integer.

Integer (int value)
Integer (String s)

byte byteValue ()
short shortValue ()
int intValue ()
long longValue ()
float floatValue ()
double doubleValue ()

static int compare (int x, int y)
int compareTo (Integer x)

static Integer decode (String s)
static int parseInt (String s)
static Integer valueOf (int i)
static Integer valueOf (String s)
String toString ()
static String toString (int i)

static String toBinaryString (int i)
static String toHexString (int i)

6.5. COMPOSICIÓN.

La composición es el agrupamiento de uno o varios objetos y valores, como atributos, que conforman el valor de los distintos objetos de una clase.

Normalmente los atributos contenidos se declaran con acceso privado y se inicializan en el constructor de la nueva clase.

La composición crea una relación "tiene", que puede ser una relación de tipo "1 a 1" si sólo se define un objeto como atributo o, una relación de tipo "1 a muchos" si se define una colección de objetos como atributos.

6.5. COMPOSICIÓN.

```
/*public*/ class Alumno {

    private String dni;
    private String nombre;

    public Alumno() {
        dni="";
        nombre="";
    }

    public String getDni() { return dni; }
    public void setDni(String dni) { this.dni=dni; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre=nombre; }
}
```

6.5. COMPOSICIÓN.

```
/*public*/ class Expediente1 {

    private String codigo;
    private String dni; /* Alumno */

    public Expediente1() {
        codigo="";
        dni="";
    }

    public String getCodigo() { return codigo; }
    public void setCodigo(String codigo) { this.codigo=codigo; }
    public String getDni() { return dni; }
    public void setDni(String dni) { this.dni=dni; }

    public Alumno getAlumno(Alumno[] t) {
        for (int i=0; i<t.length; i++)
            if (t[i]!=null)
                if (t[i].getDni().equals(this.dni))
                    return t[i];
        return null;
    }
}
```

6.5. COMPOSICIÓN.

```
/* EJEMPLO SIN COMPOSICIÓN */

Expediente1[] listaE=new Expediente1[10];
Alumno[] listaA=new Alumno[10];

Alumno a=new Alumno();
a.setDni("1");
a.setNombre("pepe");
listaA[0]=a;

Expediente1 e1=new Expediente1();
e1.setCodigo("1");
e1.setDni("1");
listaE[0]=e1;

for (int i=0; i<listaE.length; i++)
    if (listaE[i]!=null)
        System.out.printf("Codigo: %s DNI: %s Nombre: %s\n",
                          listaE[i].getCodigo(),
                          listaE[i].getDni(),
                          listaE[i].getAlumno(listaA).getNombre());
```

6.5. COMPOSICIÓN.

```java
/*public*/ class Expediente2 {

    private String codigo;
    private Alumno alumno;

    public Expediente2() {
        codigo="";
        alumno=new Alumno(); /* Esto podría cambiar si, por ejemplo,
                                utilizamos un constructor sobrecargado,
                                lo que afectaría tb al método setAlumno */
    }

    public String getCodigo() { return codigo; }
    public void setCodigo(String codigo) { this.codigo=codigo; }
    public Alumno getAlumno() { return alumno; }
    public void setAlumno(Alumno a) { this.alumno=a; }
}
```

6.5. COMPOSICIÓN.

```
/* EJEMPLO CON COMPOSICIÓN (1 a 1) */

Expediente2[] listaE=new Expediente2[10];
Alumno[] listaA=new Alumno[10];

Alumno a=new Alumno();
a.setDni("1");
a.setNombre("pepe");
listaA[0]=a;

Expediente2 e2=new Expediente2();
e2.setCodigo("1");
e2.setAlumno(a);
listaE[0]=e2;

for (int i=0; i<listaE.length; i++)
    if (listaE[i]!=null)
        System.out.printf("Codigo: %s DNI: %s Nombre: %s\n",
                          listaE[i].getCodigo(),
                          listaE[i].getAlumno().getDni(),
                          listaE[i].getAlumno().getNombre());
```

6.5. COMPOSICIÓN.

```
/*public*/ class Expediente3 {

    private String codigo;
    private Alumno[] alumnos;
    private int numAlumno;      /* Indice para conocer la posición en el
                                    array donde hay que insertar un nuevo
                                    alumno. OJO no es static en este caso. */

    public Expediente3() {
        codigo="";
        alumnos=new Alumno[10]; /* Esto podría cambiar si, por ejemplo,
                                utilizamos un constructor sobrecargado,
                                lo que afectaría tb al método setAlumno */
        numAlumno=0;
    }

    public String getCodigo() { return codigo; }
    public void setCodigo(String codigo) { this.codigo=codigo; }
    public Alumno[] getAlumnos() { return alumnos; }
    public void setAlumno(Alumno a) { this.alumnos[numAlumno++]=a; }
}
```

6.5. COMPOSICIÓN.

```
/* EJEMPLO CON COMPOSICIÓN (1 a n) */

Expediente3[] listaE=new Expediente3[10];

Alumno a=new Alumno();
a.setDni("1");
a.setNombre("pepe");

Expediente3 e3=new Expediente3();
e3.setCodigo("1");
e3.setAlumno(a);
listaE[0]=e3;

for (int i=0; i<listaE.length; i++)
    if (listaE[i]!=null)
        for (int j=0; j<listaE[i].getAlumnos().length; j++)
            if (listaE[i].getAlumnos()[j]!=null)
                System.out.printf("Codigo: %s DNI: %s Nombre: %s\n",
                                  listaE[i].getCodigo(),
                                  listaE[i].getAlumnos()[j].getDni(),
                                  listaE[i].getAlumnos()[j].getNombre());
```

6.6. CLASES ANIDADAS.

Una clase anidada es una clase que es miembro de otra clase.

```
class externa {
    …
    class anidada {
        …
    }
    …
}
```

Dado que la clase anidada es un miembro de la clase externa, tendrá acceso a todos sus métodos y atributos (incluso privados). Además la clase anidada podrá ser *private, public o protected*.

6.6. CLASES ANIDADAS.

Existen dos tipos de clases anidadas:

- Estáticas. También llamadas clases estáticas anidadas.
- No estáticas. También llamadas clases internas.

```
class externa {
    …
    static class estaticaanidada {
        …
    }
    class interna {
        …
    }
    …
}
```

6.6. CLASES ANIDADAS.

Para instanciar una clase interna se seguirá el siguiente formato:

externa.interna objetointerno = objetoexterno.new interna();

De esta manera, hay que instanciar primero la clase externa para luego instanciar la clase interna dentro del objeto externo.

¿Cuándo utilizar clases anidadas?

Cuando la clase sólo se va a utilizar en un único lugar, en ese caso, al definir la clase como anidada puede hacer que el código sea más legible y su mantenimiento sea más sencillo.

También se incrementa la encapsulación dado que la clase anidada sólo se necesita en la clase externa y de esta manera se mantienen juntas.

6.6. CLASES ANIDADAS.

```
public class Test11_clasesanidadas {

    public static void main(String[] args) {

        Alumno a=new Alumno();
        a.dir=a.new Direccion();
        a.dir.calle="";

    }
}

/*public*/ class Alumno {

    public String dni;
    public String nombre;
    public Direccion dir;

    public Alumno() {
        dni="";
        nombre="";
        //dir=new Direccion();  // Esta es otra opción de instaciación de este atributo!
    }

    public class Direccion {
        public String calle;
        public String ciudad;
    }

    /* Está todo público por simplificar el ejemplo */
}
```

FIN