

TEMA9. PROGRAMACIÓN ORIENTADA A OBJETOS (II)

**Programación
CFGS DAW**

Profesores:

Carlos Cacho

Raquel Torres

carlos.cacho@ceedcv.es

raquel.torres@ceedcv.es

Versión:180214.2038

Licencia



Reconocimiento - NoComercial - CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

ÍNDICE DE CONTENIDO

1. La clase ArrayList.....	4
1.1 Introducción.....	4
1.2 Declaración.....	4
1.3 Llenado.....	4
1.4 Acceso.....	5
1.5 Métodos.....	5
1.6 Iterando sobre un ArrayList.....	5
2. Herencia.....	6
2.1 Introducción.....	6
2.2 Constructores de clases derivadas.....	7
2.3 Métodos heredados y sobrescritos.....	8
2.4 Acceso a miembros derivados.....	8
3. Polimorfismo.....	9
4. Clases Abstractas.....	10
5. Interfaces.....	11
6. Ejemplos.....	12
6.1 Ejemplo 1.....	12
6.2 Ejemplo 2.....	14
6.3 Ejemplo 3.....	17
6.4 Ejemplo 4.....	22
6.5 Ejemplo 5.....	24
7. Agradecimientos.....	26

UD09. PROGRAMACIÓN ORIENTADA A OBJETOS (II)

1. LA CLASE ARRAYLIST

1.1 Introducción

Un *ArrayList* es una lista de tamaño variable. Son similares a los *arrays*, pero el número de elementos que puede contener no es necesario fijarlo en la creación y puede modificarse según sea necesario:

- La capacidad inicial predeterminada de un objeto *ArrayList* es 10.
- Conforme se agregan elementos a un objeto *ArrayList*, la capacidad aumenta automáticamente según lo requiera la reasignación.
- *ArrayList* crea un *array* que crece dinámicamente.

1.2 Declaración

Un *ArrayList* se declara como una clase más:

```
ArrayList lista = new ArrayList();
```

Necesitaremos importar la clase:

```
import java.util.ArrayList;
```

1.3 Llenado

Para cargar datos o elementos a una lista puede utilizarse el método *add()*.

En el siguiente ejemplo rellenamos una lista con personas:

```
lista.add(new Persona(33,21,"Luis","Maida"));
lista.add(new Persona(45,21,"Juan","Garcia"));
lista.add(new Persona(67,23,"Pedro","Poveda"));
```

Este código añade punteros a tres objetos del tipo *Persona* al *ArrayList*. Es decir, cada posición de la lista apuntará a un objeto *Persona* diferente. También se podría haber hecho lo siguiente:

```
Persona p = new Persona(33,21,"Luis","Maida");
lista.add(p);
```

1.4 Acceso

Para acceder a los elementos de una lista puede utilizarse un índice numérico que indique la posición del elemento (pero no se puede acceder a posiciones donde no se haya almacenado antes algún dato).

El *ArrayList* da a cada elemento un número índice, que hace referencia a la posición donde se encuentra el elemento. Así, el primer elemento se encuentra almacenado en el índice 0, el segundo en el 1, el tercero en el 2 y así sucesivamente.

El *ArrayList* responde a dos métodos que permiten acceder o mirar los elementos individualmente:

- El método **size()**, retorna un valor entero que indica el **número** de elementos actuales almacenados en el *ArrayList*.
- El método **get()**, tiene como argumento un número índice y retorna un **puntero** a ese índice.

1.5 Métodos

Los métodos más utilizados son:

- **void clear();** elimina todos los elementos de la lista, establece el tamaño en cero.
- **boolean isEmpty();** retorna True si la lista no contiene elementos.
- **void set(int indice, Object obj);** reemplaza el puntero en el índice dado de la lista. Es el opuesto del método *get()*.
- **boolean contains(Object obj);** busca en la lista y retorna True si contiene el objeto dado. Usa el método *equals()* para comparar objetos.
- **int indexOf(Object obj);** similar a *contains()*, busca en la lista desde la posición 0..n-1 el objeto dado. Retorna la posición o índice dónde se encuentra el objeto, o -1 si no lo encuentra. Usa *equals()* para comparar los objetos.
- **boolean remove(Object obj);** busca en la lista el objeto y si lo encuentra lo elimina de la lista. Luego, decrementa el índice en uno. Retorna True si el objeto es encontrado y eliminado, False en otro caso. Usa *equals()* para comparar los objetos.
- **void remove(int index);** similar a *remove()* anterior, pero toma el índice del elemento que se desea eliminar. Este método es más rápido que el otro, puesto que no necesita recorrer cada elemento de la lista.

1.6 Iterando sobre un ArrayList

Hay dos maneras diferentes en las que se puede iterar (repetir) sobre una colección de tipo *ArrayList*.

- Utilizando el bucle *for* y el método *get()* con un índice.
- Usando el objeto *Iterator*. Es un objeto que nos permite recorrer listas

como si fuese un índice que la recorre.

El objeto *Iterator* comúnmente maneja dos métodos:

- *hasNext()*: Verifica si hay más elementos
- *next()*: recoge el objeto y avanza.

Para utilizarlo se necesita importar la clase:

```
import java.util.Iterator;
```

Utilizando el bucle for:

```
for(int i = 0; i < lista.size(); i++) {  
    System.out.println(lista.get(i).toString()); //get(i) devuelve un  
    puntero al elemento i                        del array, por  
    tanto necesitamos utilizar                    el  
    método toString() para poder sacarlo  
    por pantalla.  
}
```

Utilizando el objeto *Iterator*:

```
Iterator iter = lista.iterator(); // Creamos el objeto a partir de la lista  
while(iter.hasNext() && encontrado == false ) // Mientras haya siguiente  
en la lista  
{  
    if( iter.next().toString().equalsIgnoreCase( p.toString() ) ) //  
Obtenemos el elemento  
    encontrado = true;  
}
```

Ver [ejemplo 1](#).

Ver [ejemplo 2](#).

2. HERENCIA

2.1 Introducción

La herencia es una de las capacidades más importantes y distintivas de la POO. La declaración de la herencia se hace a través de la palabra reservada *extends*.



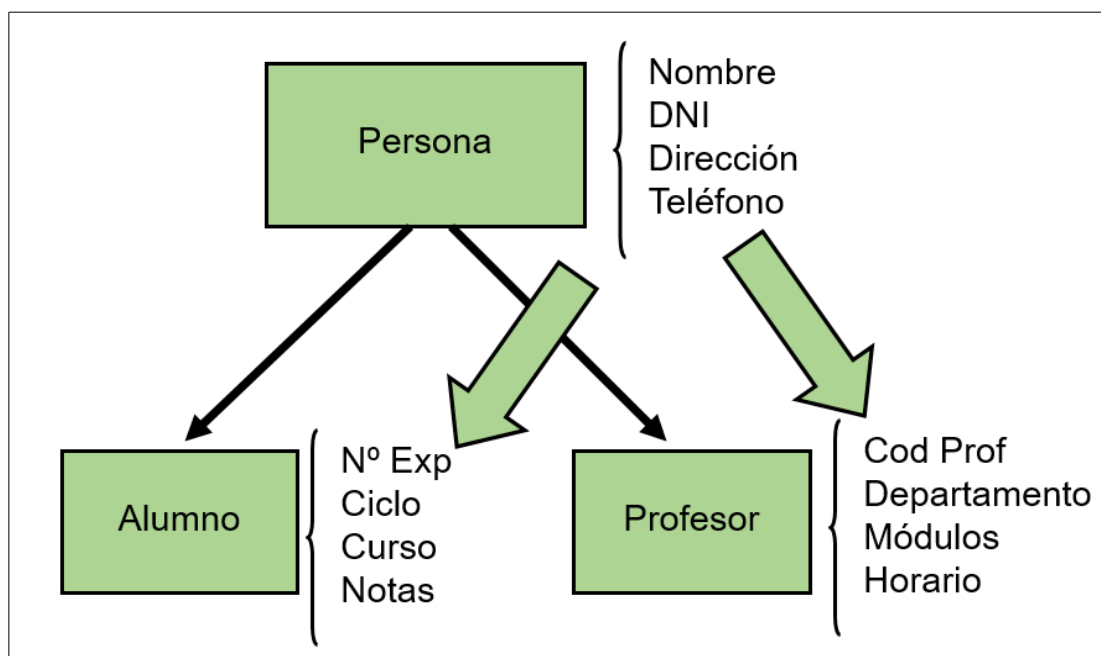
Quando derivamos (o extendemos) una nueva clase, ésta hereda todos los datos y métodos miembro de la clase existente.

De esta manera, si no se introduce ningún nuevo atributo ni ningún nuevo método, la nueva clase es exactamente igual que la clase de la que deriva.

Por ejemplo, si tenemos un programa que va a trabajar con alumnos y profesores, éstos van a compartir atributos tales como el nombre, dni, dirección o teléfono. Pero cada uno de ellos tendrán atributos que no tengan los otros, los alumnos tendrán el número de expediente, el ciclo y curso que cursan y sus notas. Por su parte los profesores tendrán el código de profesor, el departamento al que pertenecen, los módulos que imparten y su horario.

Por lo tanto, en este caso lo mejor es crear una clase *Persona* con los atributos que compartirán tanto los alumnos como los profesores y heredar de ésta dos clases con sus atributos propios como son *Alumno* y *Profesor*.

Es importante recalcar que no sólo heredarán los atributos sino también los métodos de la clase madre (*Persona*).



Para declarar una clase derivada se utiliza la sintaxis:

```
class nombre_clase_derivada extends nombre_clase_base {  
    ...  
}
```

2.2 Constructores de clases derivadas

El constructor de una clase derivada debe construir, por un lado, una parte que está definida en la clase base y, por otro lado, los nuevos atributos que se han añadido a la clase derivada.

La llamada al constructor de la clase base se realiza a través de la palabra

reservada **super**, y se lleva a cabo dentro de la implementación del constructor derivado.

En el caso de que un constructor no inicialice explícitamente la clase base, el compilador llama automáticamente al constructor por defecto de la clase base; si ésta no tiene un constructor por defecto, el compilador generará un error.

Cuando se crea una instancia de una clase derivada, el compilador llama a los constructores por el siguiente orden:

1. Llamada al constructor de la clase base.
2. Constructores de los atributos en el orden en que han sido definidos.
3. Ejecución de las líneas de código del constructor de la clase derivada.
4. Llamada al constructor correspondiente a cada objeto (para aquellos objetos que no hayan sido contruidos en la definición de la clase), cada vez que se utilice la palabra *new* en el código del constructor.

2.3 Métodos heredados y sobreescritos

Hemos visto que una clase derivada consta de los mismos atributos y métodos que la clase base de la que deriva; además, se pueden incluir nuevos atributos y nuevos métodos. Puede ocurrir que alguno de los métodos que existen en la clase base no nos sirvan tal y como están programados y, por lo tanto, necesitemos adecuarlos a la nueva clase derivada.

Podremos hablar de dos tipos de métodos heredados:

- **Métodos heredados:** un método es heredado cuando en la clase derivada se puede utilizar de la misma manera que en la clase base. Por ejemplo, en una clase dada *Alumno* que deriva de una clase *Persona*, podríamos reutilizar un método *getNombre()*, puesto que no necesitamos ninguna información específica añadida al pasar a la clase *Alumno*.
- **Métodos sobreescritos:** un método es sobreescrito o está reimplementado, cuando debe ser programado de nuevo para que ejecute un conjunto de instrucciones que tengan el mismo sentido para dos clases diferentes. Por ejemplo el método *mostrarPersona()* en el que debe mostrarse la totalidad de los atributos del objeto.

El método sobreescrito podría reutilizar el método de la clase base y, a continuación, añadir los nuevos atributos. En Java, podemos acceder a un método definido en la clase base que se haya sobreescrito empleando de nuevo la palabra reservada *super*:

```
super.mostrarPersona();
```

Ver [ejemplo 3](#).

2.4 Acceso a miembros derivados

Aunque una subclase incluye todos los miembros de su superclase o clase madre, no puede acceder a aquellos miembros de las superclases que hayan sido declarados como *private*.

De hecho si en el ejemplo 3 intentásemos acceder desde las clases derivadas a los atributos de la clase *Persona* (que son privados) obtendríamos un error ya que no pueden ser accedidos desde fuera de la clase.

También podemos declarar los atributos como *protected*. De esta forma solo podrán ser accedidos desde las clases heredadas, nunca desde otras clases.



Los atributos declarados como *protected* son públicos para las clases heredadas pero privadas para las demás clases.

Los distintos niveles de acceso en la herencia son:

Objetos	public	private	protected
Misma clase	SÍ	SÍ	SÍ
Clase heredada del mismo paquete	SÍ	NO	SÍ
Clase no heredada del mismo paquete	SÍ	NO	SÍ
Clase heredada de diferente paquete	SÍ	NO	SÍ
Clase no heredada de diferente paquete	SÍ	NO	NO

3. POLIMORFISMO

La sobreescritura de métodos constituye la base de uno de los conceptos más potentes de Java: la **selección dinámica de métodos**, que es un mecanismo mediante el cual la llamada a un método sobreescrito se resuelve durante el tiempo de ejecución y no en el de compilación. La selección dinámica de métodos es importante porque permite implementar el polimorfismo durante el tiempo de ejecución.

Una variable de referencia a una superclase se puede referir a un objeto de una subclase. Java se basa en esto para resolver llamadas a métodos sobreescritos en el tiempo de ejecución.

Lo que determina la versión del **método** que será **ejecutado** es **el tipo de objeto** al que se hace referencia y no el tipo de variable de referencia.

El polimorfismo es básico en la programación orientada a objetos porque

permite que una clase general especifique métodos que serán comunes a todas las clases que se deriven de esa misma clase, de esta manera las subclases podrán definir la implementación de alguno o de todos esos métodos.

La superclase proporciona todos los elementos que una subclase puede usar directamente. También define aquellos métodos que las subclases que se deriven de ella deben implementar por sí mismas.

De esta manera, combinando la herencia y la sobreescritura de métodos, una superclase puede definir la forma general de los métodos que se usarán en todas sus subclases

Ver [ejemplo 4](#).

4. CLASES ABSTRACTAS

A veces se quiere crear una superclase que únicamente defina una forma generalizada que se compartirá con todas las subclases, dejando que cada subclase que complemente los detalles necesarios. Una clase de este tipo determina la naturaleza de los métodos que las subclases han de implementar.

Se pueden definir métodos que se deberán sobrecribir en la subclase para tener un significado completo. Pero hay que asegurarse de que una subclase sobrecribe perfectamente todos los métodos necesarios. La solución que ofrece Java a este problema es el método abstracto.

Puede ser necesario que las subclases sobrecriban ciertos métodos especificando el modificador de tipo *abstract*. Para declarar un método abstracto se utiliza esta expresión general:

```
abstract tipo_nombre(lista_de_parametros);
```

✎ Cualquier clase que contenga un método abstracto o más debe declararse como abstracta.

Para declarar una superclase abstracta basta con anteponer la palabra *abstract* delante de la palabra clave *class* al principio de la declaración de la clase.

✎ No puede haber objetos de una clase abstracta, es decir, no se pueden crear instancias de dichas clases directamente con el operador *new*.

Tales objetos no tendrían ninguna utilidad puesto que no esta totalmente definida.

⚡ No pueden declararse constructores abstractos, o métodos estáticos abstractos.

Cualquier subclase de una clase abstracta ha de implementar todos los métodos abstractos de la superclase, o declararse a sí misma como abstracta.

⚡ Algunas veces es necesario evitar que una clase sea heredada. Para evitar hacerlo, hay que anteponer la palabra clave *final* al nombre de la clase.

Declarando una clase como *final*, implícitamente, se declaran todos sus métodos como *final*, como podemos imaginar es imposible declarar una clase como *abstract* y *final* al mismo tiempo.

5. INTERFACES

La programación orientada a objetos es una programación orientada a la interfaz que pretende ocultar la implementación. Podemos heredar de una clase existente, lo cual implica que la nueva clase asume y hace propia la interfaz de la clase de la cual deriva.

Mediante la construcción de un interfaz, el programador pretende especificar qué comportamiento caracteriza a una colección de objetos e, igualmente, especificar qué comportamiento deben reunir los objetos que quieran entrar dentro de sea categoría o colección.

📖 Un interfaz en Java es una declaración de métodos abstractos sin implementación.

Aunque también se pueden declarar constantes que definen el comportamiento que deben soportar los objetos que quieran implementar esa interfaz.

La sintaxis de una interfaz es la siguiente:

```
Modificador interface Nombre {  
    // Declaración de los métodos y atributos que constituyen la  
    // interfaz y definen el comportamiento común a las clases que  
    // vayan a implementarla.  
}
```

Si una interfaz define un tipo (al igual que una clase define un tipo) pero ese tipo no provee de ningún método, podemos preguntarnos: ¿para qué sirven entonces las interfaces en Java?

La implementación (herencia) de una interfaz no podemos decir que evite la duplicidad de código o que favorezca la reutilización de código puesto que realmente no proveen código.

En cambio sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque **al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.**

Un aspecto fundamental de las interfaces en Java es **separar la especificación de una clase (qué hace) de la implementación (cómo lo hace)**. Esto se ha comprobado que da lugar a programas más robustos y con menos errores.

Es importante tener en cuenta que:

- Una clase puede implementar varias interfaces (separadas por comas)
- Una clase que implementa una interfaz debe de proporcionar implementación para todos y cada uno de los métodos definidos en la misma.
- Las clases que implementan una interfaz que tiene definidas constantes pueden usarlas en cualquier parte del código de la clase, simplemente indicando su nombre.

Si por ejemplo la clase *Alumno* implementa la interfaz *Interfaz* la sintaxis sería:

```
public class Alumno implements Interfaz {  
}
```

Ver [ejemplo 5](#).

6. EJEMPLOS

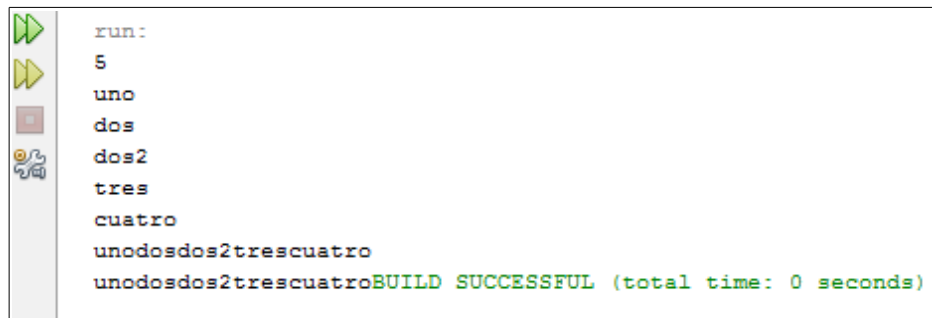
6.1 Ejemplo 1

Ejemplo que crea, rellena y recorre un *ArrayList*.

(Nótese que cuando hacemos *l.get(i).toString* estamos obteniendo el *String* que contiene el objeto *String* de la posición *i* y no la conversión a *String* de la posición de memoria del objeto, porque la clase *String* (tipo de los elemento que contiene este *ArrayList*) ya implementa el método *.toString* que devuelve el valor del *String* que contiene. Además cabe destacar, que por defecto, el método *System.out.println()* invoca al método *toString()* de nuestro objeto).

```
8  import java.util.ArrayList;
9  import java.util.Iterator;
10
11  public class Ejemplos {
12
13      public static void main(String[] args) {
14
15          // Creamos la lista
16          ArrayList l = new ArrayList() ;
17
18          // Añadimos elementos al final de la lista
19          l.add("uno");
20          l.add("dos");
21          l.add("tres");
22          l.add("cuatro");
23
24          // Añadimos el elemento en la posición 2
25          l.add(2, "dos2");
26
27          System.out.println(l.size()); // Devuelve 5
28          System.out.println(l.get(0)); // Devuelve uno
29          System.out.println(l.get(1)); // Devuelve dos
30          System.out.println(l.get(2)); // Devuelve dos2
31          System.out.println(l.get(3)); // Devuelve tres
32          System.out.println(l.get(4)); // Devuelve cuatro
33
34          //Recorremos la lista con un for y mostramos el contenido
35          for (int i=0; i<l.size(); i++) {
36              // DEbemos usar el método toString para pasar el objeto a String
37              System.out.print(l.get(i).toString());
38          } // Imprime: unodosdos2trescuatro
39
40          System.out.print("\n");
41
42          // Recorremos la lista con un iterador
43          // Creamos el iterador
44          Iterator it = l.iterator();
45
46          // Mientras hayan elementos
47          while(it.hasNext()) {
48              System.out.print(it.next().toString()); // Obtengo el elemento
49          } // Imprime: unodosdos2trescuatro
50      }
51  }
```

Salida:



```
run:
5
uno
dos
dos2
tres
cuatro
unodosdos2trescuatro
unodosdos2trescuatroBUILD SUCCESSFUL (total time: 0 seconds)
```

6.2 Ejemplo 2

En este ejemplo vamos a crear una clase Producto con dos atributos:

- nombre: String
- cantidad: int

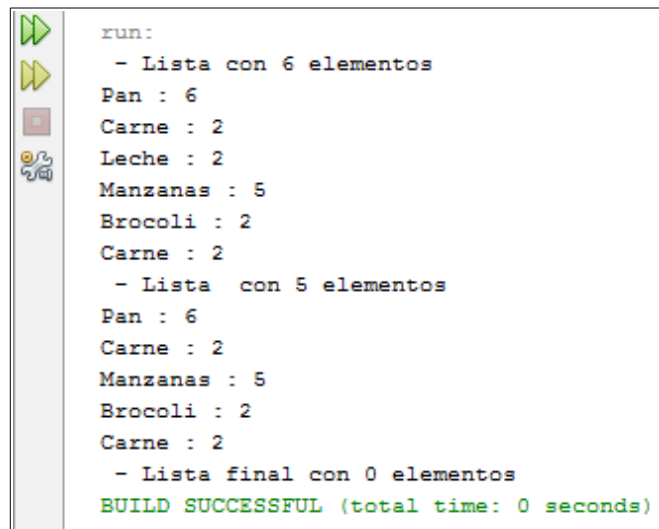
Crearemos un constructor sin parámetros y otro con parámetros y los métodos get y set asociados a los atributos.

```
9 public class Producto {
10     // Atributos
11     private String nombre;
12     private int cantidad;
13
14     // Métodos
15
16     // Constructor con parámetros donde asignamos el valor dado a los atributos
17     public Producto(String nom, int cant){
18         this.nombre = nom;
19         this.cantidad = cant;
20     }
21
22     // Constructor sin parámetros donde inicializamos los atributos
23     public Producto() {
24         // La palabra reservada null se utiliza para inicializar los objetos,
25         // indicando que el puntero del objeto no apunta a ninguna dirección
26         // de memoria. No hay que olvidar que String es una clase.
27         this.nombre = null;
28         this.cantidad = 0;
29     }
30
31     // Métodos get y set
32     public String getNombre() {
33         return nombre;
34     }
35
36     public void setNombre(String nombre) {
37         this.nombre = nombre;
38     }
39
40     public int getCantidad() {
41         return cantidad;
42     }
43
44     public void setCantidad(int cantidad) {
45         this.cantidad = cantidad;
46     }
47 }
48
```

A continuación en el programa principal crearemos una lista de productos y haremos operaciones sobre ella.

```
8  import java.util.ArrayList;
9  import java.util.Iterator;
10
11  public class Ejemplos {
12
13      public static void main(String[] args) {
14
15          // Definimos 5 instancias de la Clase Producto
16          Producto p1 = new Producto("Pan", 6);
17          Producto p2 = new Producto("Leche", 2);
18          Producto p3 = new Producto("Manzanas", 5);
19          Producto p4 = new Producto("Brocoli", 2);
20          Producto p5 = new Producto("Carne", 2);
21
22          // Definir un ArrayList
23          ArrayList lista = new ArrayList();
24
25          // Colocar Instancias de Producto en ArrayList
26          lista.add(p1);
27          lista.add(p2);
28          lista.add(p3);
29          lista.add(p4);
30
31          // Añadimos "Carne" en la posición 1 de la lista
32          lista.add(1, p5);
33
34          // Añadimos "Carne" en la última posición
35          lista.add(p5);
36
37          // Imprimir contenido de ArrayLists
38          System.out.println(" - Lista con " + lista.size() + " elementos");
39
40          // Definir Iterator para extraer/imprimir valores
41          // si queremos utilizar un for con el iterador no hace falta poner el incremento
42          for( Iterator it = lista.iterator(); it.hasNext(); )
43          {
44              // Hacemos un casting para poder guardarlo en una variable Producto
45              Producto p = (Producto)it.next();
46              System.out.println(p.getNombre() + " : " + p.getCantidad());
47          }
48
49          // Eliminar elemento de ArrayList
50          lista.remove(2);
51          System.out.println(" - Lista con " + lista.size() + " elementos");
52
53          // Definir Iterator para extraer/imprimir valores
54          for( Iterator it2 = lista.iterator(); it2.hasNext(); ) {
55              Producto p = (Producto)it2.next();
56              System.out.println(p.getNombre() + " : " + p.getCantidad());
57          }
58
59          // Eliminar todos los valores del ArrayList
60          lista.clear();
61          System.out.println(" - Lista final con " + lista.size() + " elementos");
62      }
63  }
```


Salida:



```
run:
- Lista con 6 elementos
Pan : 6
Carne : 2
Leche : 2
Manzanas : 5
Brocoli : 2
Carne : 2
- Lista con 5 elementos
Pan : 6
Carne : 2
Manzanas : 5
Brocoli : 2
Carne : 2
- Lista final con 0 elementos
BUILD SUCCESSFUL (total time: 0 seconds)
```

[volver](#)

6.3 Ejemplo 3

En este ejemplo vamos a crear la clase Persona y sus clases heredadas: Alumno y Profesor. En la clase Persona crearemos el constructor, un método para mostrar los atributos y los métodos get y set asociados a cada atributo. Las clases Alumno y Profesor heredarán de la clase Persona (utilizando la palabra reservada `extends`) y cada una tendrá sus propios atributos, un constructor que llamará también al constructor de la clase Persona (utilizando el método `super()`), un método para mostrar sus atributos, que también llamará al método de Persona y los métodos get y set asociados a sus atributos.

Es interesante ver cómo se ha sobrescrito el método `mostrarPersona()` en las clases heredadas. El método se llama igual y hace uso de la palabra reservada `super` para llamar al método de `mostrarPersona()` de la clase Persona. En la llamada del main tanto el objeto `a` (Alumno) como el objeto `profe` (Profesor) pueden hacer uso del método `mostrarPersona()`.

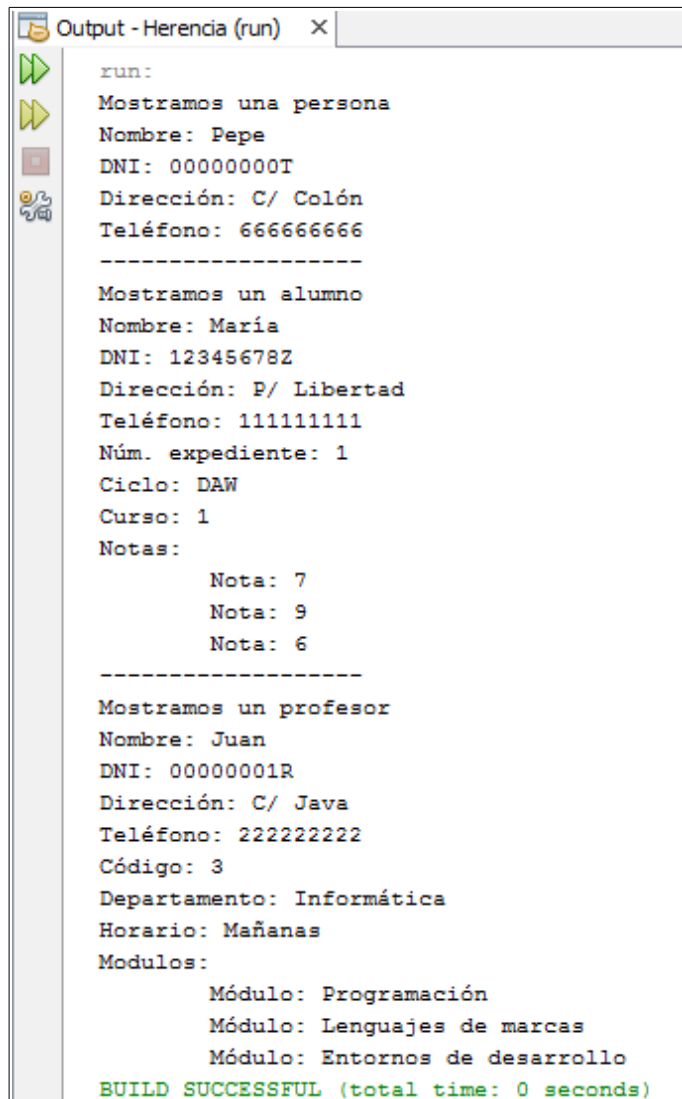
```
public class Persona {  
10  
11     private String nombre;  
12     private String dni;  
13     private String direccion;  
14     private int telefono;  
15  
16     public Persona(String nom, String dni, String direc, int tel)  
17     {  
18         this.nombre = nom;  
19         this.dni = dni;  
20         this.direccion = direc;  
21         this.telefono = tel;  
22     }  
23  
24     public void mostrarPersona()  
25     {  
26         System.out.println("Nombre: " + this.nombre);  
27         System.out.println("DNI: " + this.dni);  
28         System.out.println("Dirección: " + this.direccion);  
29         System.out.println("Teléfono: " + this.telefono);  
30     }  
31  
32     public String getNombre() {  
33         return nombre;  
34     }  
35  
36     public void setNombre(String nombre) {  
37         this.nombre = nombre;  
38     }  
39  
40     public String getDni() {  
41         return dni;  
42     }  
43  
44     public void setDni(String dni) {  
45         this.dni = dni;  
46     }  
47  
48     public String getDireccion() {  
49         return direccion;  
50     }  
51  
52     public void setDireccion(String direccion) {  
53         this.direccion = direccion;  
54     }  
55  
56     public int getTelefono() {  
57         return telefono;  
58     }  
59  
60     public void setTelefono(int telefono) {  
61         this.telefono = telefono;  
62     }  
63  
64 }
```

```
8  import java.util.ArrayList;
9  import java.util.Iterator;
10
11
12  public class Alumno extends Persona{
13
14      private int exp;
15      private String ciclo;
16      private int curso;
17      private ArrayList notas;
18
19      // Al constructor hemos de pasarle los atributos de la clase Alumno y la de Persona
20      public Alumno(String nom, String dni, String direc, int tel, int exp, String ciclo, int curso, ArrayList notas)
21      {
22          // Llamamos al constructor de la clase Persona
23          super(nom, dni, direc, tel);
24
25          this.exp = exp;
26          this.ciclo = ciclo;
27          this.curso = curso;
28          this.notas = notas;
29      }
30
31      public void mostrarPersona()
32      {
33          // Llamamos al método de la clase madre para que muestre los datos de Persona
34          super.mostrarPersona();
35
36          System.out.println("Núm. expediente: " + this.exp);
37          System.out.println("Ciclo: " + this.ciclo);
38          System.out.println("Curso: " + this.curso);
39          System.out.println("Notas:");
40          for( Iterator it = this.notas.iterator(); it.hasNext(); )
41          {
42              System.out.println("\tNota: " + it.next());
43          }
44      }
45
46      public int getExp() {
47          return exp;
48      }
49
50      public void setExp(int exp) {
51          this.exp = exp;
52      }
53
54      public String getCiclo() {
55          return ciclo;
56      }
57
58      public void setCiclo(String ciclo) {
59          this.ciclo = ciclo;
60      }
61
62      public int getCurso() {
63          return curso;
64      }
65
66      public void setCurso(int curso) {
67          this.curso = curso;
68      }
69
70      public ArrayList getNotas() {
71          return notas;
72      }
73
74      public void setNotas(ArrayList notas) {
75          this.notas = notas;
76      }
77  }
```

```
8  import java.util.ArrayList;
9  import java.util.Iterator;
10
11  public class Profesor extends Persona{
12
13      private int cod;
14      private String depto;
15      private ArrayList modulos;
16      private String horario;
17
18      // Al constructor hemos de pasarle los atributos de la clase Peofesor y la de Persona
19      public Profesor(String nom, String dni, String direc, int tel, int cod, String depto, ArrayList mod, String horario)
20      {
21          // Llamamos al constructor de la clase Persona
22          super(nom, dni, direc, tel);
23
24          this.cod = cod;
25          this.depto = depto;
26          this.modulos = mod;
27          this.horario = horario;
28      }
29      public void mostrarPersona()
30      {
31          // Llamamos al método de la clase madre para que muestre los datos de Persona
32          super.mostrarPersona();
33
34          System.out.println("Código: " + this.cod);
35          System.out.println("Departamento: " + this.depto);
36          System.out.println("Horario: " + this.horario);
37          System.out.println("Modulos:");
38          for( Iterator it = this.modulos.iterator(); it.hasNext(); )
39          {
40              System.out.println("\tMódulo: " + it.next());
41          }
42      }
43
44      public int getCod() {
45          return cod;
46      }
47
48      public void setCod(int cod) {
49          this.cod = cod;
50      }
51
52      public String getDepto() {
53          return depto;
54      }
55
56      public void setDepto(String depto) {
57          this.depto = depto;
58      }
59
60      public ArrayList getModulos() {
61          return modulos;
62      }
63
64      public void setModulos(ArrayList modulos) {
65          this.modulos = modulos;
66      }
67
68      public String getHorario() {
69          return horario;
70      }
71
72      public void setHorario(String horario) {
73          this.horario = horario;
74      }
75  }
```

```
8  import java.util.ArrayList;
9
10 public class Herencia {
11
12     public static void main(String[] args) {
13
14         // Probamos la clase persona
15         // Llamamos al constructor con el nombre, dni, dirección y teléfono
16         Persona p = new Persona("Pepe", "00000000T", "C/ Colón", 666666666);
17
18         System.out.println("Mostramos una persona");
19         p.mostrarPersona();
20
21         //Probamos la clase Alumno
22
23         // Creamos las notas
24         ArrayList notas = new ArrayList();
25
26         notas.add(7);
27         notas.add(9);
28         notas.add(6);
29
30         // Llamamos al constructor con el nombre, dni, dirección, teléfono, expediente, ciclo, curso y las notas
31         Alumno a = new Alumno("María", "12345678Z", "P/ Libertad", 11111111, 1, "DAW", 1, notas);
32
33         System.out.println("-----");
34         System.out.println("Mostramos un alumno");
35         a.mostrarPersona();
36
37         //Probamos la clase Profesor
38
39         // Creamos los módulos
40         ArrayList modulos = new ArrayList();
41
42         modulos.add("Programación");
43         modulos.add("Lenguajes de marcas");
44         modulos.add("Entornos de desarrollo");
45
46         // Llamamos al constructor con el nombre, dni, dirección, teléfono, expediente, ciclo, curso y las notas
47         Profesor profe = new Profesor("Juan", "00000001R", "C/ Java", 22222222, 3, "Informática", modulos, "Mañanas");
48
49         System.out.println("-----");
50         System.out.println("Mostramos un profesor");
51
52         profe.mostrarPersona();
53     }
54 }
55 }
```

Salida:



```
run:
Mostramos una persona
Nombre: Pepe
DNI: 00000000T
Dirección: C/ Colón
Teléfono: 666666666
-----
Mostramos un alumno
Nombre: María
DNI: 12345678Z
Dirección: P/ Libertad
Teléfono: 111111111
Núm. expediente: 1
Ciclo: DAW
Curso: 1
Notas:
    Nota: 7
    Nota: 9
    Nota: 6
-----
Mostramos un profesor
Nombre: Juan
DNI: 00000001R
Dirección: C/ Java
Teléfono: 222222222
Código: 3
Departamento: Informática
Horario: Mañanas
Modulos:
    Módulo: Programación
    Módulo: Lenguajes de marcas
    Módulo: Entornos de desarrollo
BUILD SUCCESSFUL (total time: 0 seconds)
```

[volver](#)

6.4 Ejemplo 4

En este ejemplo vamos a trabajar el polimorfismo. Para ello vamos a crear una clase Madre con un método llamame(). A continuación crearemos dos clases derivadas de ésta: Hija1 e Hija2, sobrescribiendo el método llamame().

En el main crearemos un objeto de cada clase y los asignaremos a una variable de tipo Madre (madre2) con la que llamaremos al método. Es interesante observar que al asignar la variable madre2 a cada uno de los objetos obtiene la referencia del mismo. Es decir si la igualamos a un objeto Hija1 el método utilizado es el de la clase Hija1 y si la igualamos a un objeto Hija2 el método será el de la clase Hija2.

```
class Madre {
    void llamame(){
        System.out.println("Estoy en la clase Madre");
    }
}

class Hija1 extends Madre {
    void llamame(){
        System.out.println("Estoy en la subclase Hija1");
    }
}

class Hija2 extends Madre {
    void llamame(){
        System.out.println("Estoy en la subclase Hija2");
    }
}

class Ejemplo {
    public static void main(String args[]){
        // Creamos un objeto de cada clase
        Madre madre = new Madre();
        Hija1 h1 = new Hija1();
        Hija2 h2 = new Hija2();

        // Declaramos otra variable de tipo Madre
        Madre madre2;

        // Asignamos a madre2 el objeto madre
        madre2 = madre;
        madre2.llamame();

        // Asignamos a madre2 el objeto h1 (Hija1)
        madre2 = h1;
        madre2.llamame();

        // Asignamos a madre2 el objeto h2 (Hija2)
        madre2 = h2;
        madre2.llamame();
    }
}
```

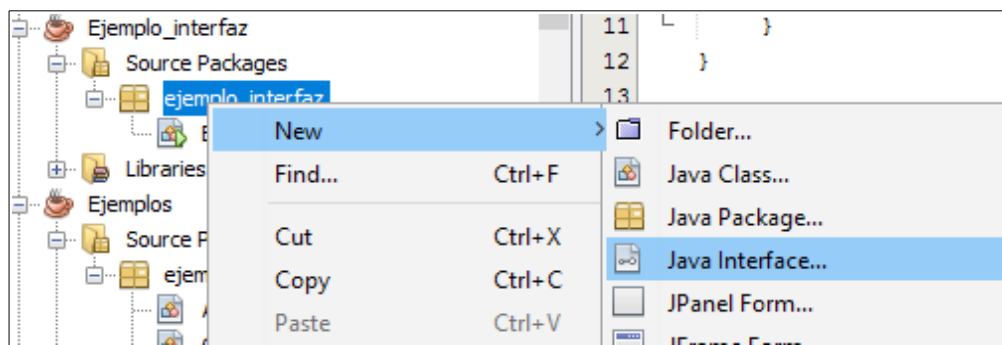
Salida:

```
run:
Estoy en la clase Madre
Estoy en la subclase Hija1
Estoy en la subclase Hija2
BUILD SUCCESSFUL (total time: 0 seconds)
```

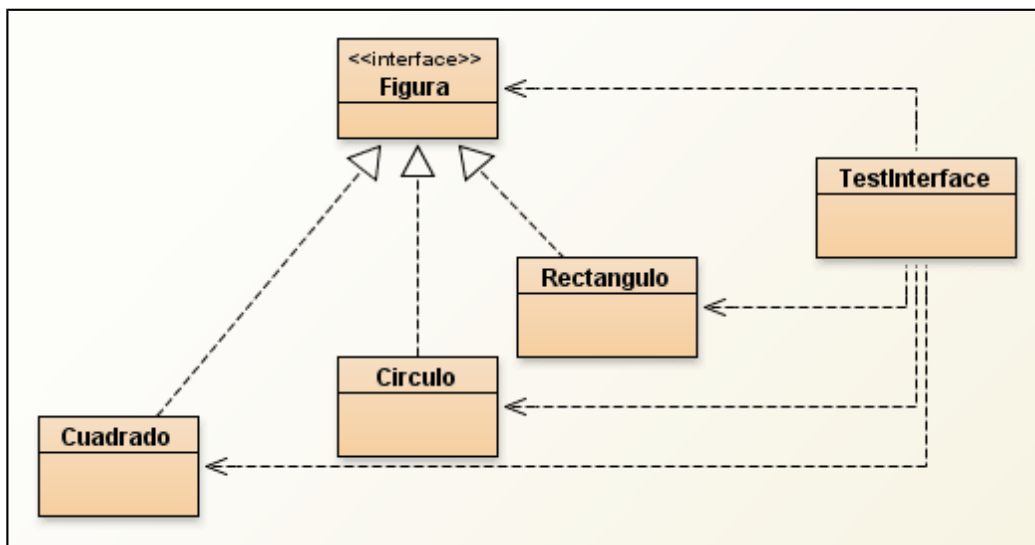
[volver](#)

6.5 Ejemplo 5

En este ejemplo vamos a crear una interfaz y posteriormente implementarla en una clase. Para crear una interfaz debemos pinchar con el botón derecho sobre el paquete donde la queramos crear y después NEW > Java Interface.



Vamos a ver un ejemplo simple de definición y uso de interfaz en Java. Las clases que vamos a usar y sus relaciones se muestran en el esquema:



```
public interface Figura {  
    float PI = 3.1416f; // Por defecto public static final. La f final indica que el número es float  
    float area(); // Por defecto abstract public  
}
```



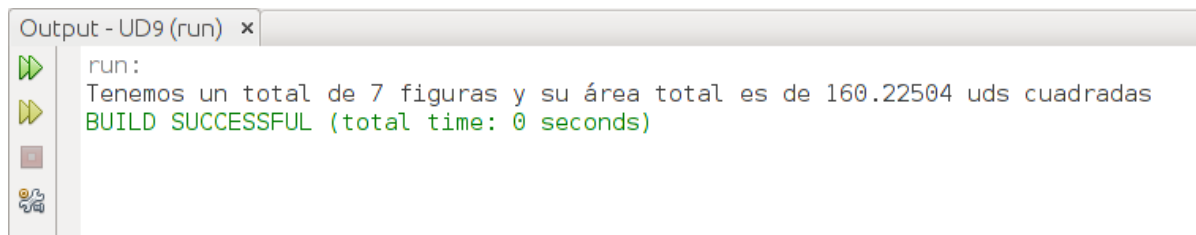
```
12 public class Cuadrado implements Figura {  
13     private float lado;  
14  
15     public Cuadrado (float lado) {  
16         this.lado = lado;  
17     }  
18  
19     public float area() {  
20         return lado*lado;  
21     }  
22 }  
23
```

```
12 public class Rectangulo implements Figura {  
13     private float lado;  
14     private float altura;  
15  
16     public Rectangulo (float lado, float altura) {  
17         this.lado = lado;  
18         this.altura = altura;  
19     }  
20  
21     public float area() {  
22         return lado*altura;  
23     }  
24 }
```

```
12 public class Circulo implements Figura {  
13     private float diametro;  
14  
15     public Circulo (float diametro) {  
16         this.diametro = diametro;  
17     }  
18  
19     public float area() {  
20         return (PI*diametro*diametro/4f);  
21     }  
22 }  
23
```

```
21 public static void main(String[] args) {
22     // TODO code application logic here
23     Figura cuad1 = new Cuadrado (3.5f);
24     Figura cuad2 = new Cuadrado (2.2f);
25     Figura cuad3 = new Cuadrado (8.9f);
26
27     Figura circ1 = new Circulo (3.5f);
28     Figura circ2 = new Circulo (4f);
29
30     Figura rect1 = new Rectangulo (2.25f, 2.55f);
31     Figura rect2 = new Rectangulo (12f, 3f);
32
33     ArrayList serieDeFiguras = new ArrayList();
34
35     serieDeFiguras.add (cuad1);
36     serieDeFiguras.add (cuad2);
37     serieDeFiguras.add (cuad3);
38
39     serieDeFiguras.add (circ1);
40     serieDeFiguras.add (circ2);
41     serieDeFiguras.add (rect1);
42     serieDeFiguras.add (rect2);
43
44     float areaTotal = 0;
45     Iterator it = serieDeFiguras.iterator(); //creamos un iterador
46
47     while (it.hasNext()){
48         Figura tmp = (Figura)it.next();
49         areaTotal = areaTotal + tmp.area();
50     }
51
52     System.out.println ("Tenemos un total de " + serieDeFiguras.size() + " figuras y su área total es de " +
53         areaTotal + " uds cuadradas");
54 }
```

El resultado de ejecución podría ser algo así:



```
Output - UD9 (run) x
run:
Tenemos un total de 7 figuras y su área total es de 160.22504 uds cuadradas
BUILD SUCCESSFUL (total time: 0 seconds)
```

En este ejemplo **comprobamos que la interface Figura define un tipo**.

Podemos crear un ArrayList de figuras donde tenemos figuras de distintos tipos (cuadrados, círculos, rectángulos) aprovechándonos del polimorfismo. Esto nos permite darle un tratamiento común a todas las figuras. En concreto, usamos un bucle *for* para recorrer la lista de figuras y obtener un área total.

7. AGRADECIMIENTOS

Apuntes actualizados y adaptados al CEEDCV a partir de la siguiente documentación:

[1] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.