


UNIDAD 7


PROGRAMACIÓN ORIENTADA A OBJETOS II


Programación
CFGS DAW

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 Métodos de Acceso y Manipulación.....	5
1.5 Recorrido de un ArrayList.....	5
1.6 Ejemplo 1.....	6
1.7 Ejemplo 2.....	7
2. Composición.....	9
3. Herencia.....	10
3.1 Introducción.....	10
3.2 Constructores de clases derivadas.....	11
3.3 Métodos heredados y sobrescritos.....	11
3.4 Clases y métodos final.....	11
3.5 Acceso a miembros derivados.....	11
3.6 Ejemplo 3.....	12
4. Polimorfismo.....	16
4.1 Ejemplo 4.....	16
5. Clases Abstractas.....	18
6. Interfaces.....	19
6.1 Ejemplo 6.....	20
7. Agradecimientos.....	22

UD09. PROGRAMACIÓN ORIENTADA A OBJETOS II

1. LA CLASE ARRAYLIST

La Clase ArrayList no está directamente relacionada con la programación orientada a objetos, pero este es un buen momento para aprender a utilizarla.

1.1 Introducción

Un *ArrayList* es una estructura de datos dinámica del tipo **colección** que implementa una lista de tamaño variable. Es similar a un *array* con las ventajas de que **su tamaño crece dinámicamente conforme se añaden elementos** (no es necesario fijar su tamaño al crearlo) y **permite almacenar datos y objetos de cualquier tipo**.

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 insertar datos o elementos en un *ArrayList* puede utilizarse el método *add()*.

En el siguiente ejemplo insertamos datos int, double, char y String:

```
lista.add(-25);  
lista.add(3.14);  
lista.add('A');  
lista.add("Luis");
```

En el siguiente ejemplo insertamos objetos de la clase *Persona*:

```
lista.add(new Persona("28751533Q", "María", "Maida García", 37));  
lista.add(new Persona("65971552A", "Luis", "González Collado", 17));  
lista.add(new Persona("16834954R", "Raquel", "Dobón Pérez", 62));
```

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("28751533Q", "María", "Maida García", 37);  
lista.add(p);
```

1.4 Métodos de Acceso y Manipulación

Un *ArrayList* da a cada elemento insertado un índice numérico que hace referencia a la posición donde se encuentra, de forma similar a un array. Así, el primer elemento se encuentra almacenado en el índice 0, el segundo en el 1, el tercero en el 2 y así sucesivamente.

Los métodos más utilizados para acceder y manipular un *ArrayList* son:

- *int size()*; devuelve el número de elementos de la lista.
- *E get(int index)*; devuelve una referencia al elemento en la posición index.
- *void clear()*; elimina todos los elementos de la lista. Establece el tamaño a cero.
- *boolean isEmpty()*; retorna true si la lista no contiene elementos.
- *boolean add(E element)*; inserta element al final de la lista y devuelve true.
- *void add(int index, E element)*; inserta element en la posición index de la lista. Desplaza una posición todos los demás elementos de la lista (no sustituye ni borra otros elementos).
- *void set(int index, E element)*; sustituye el elemento en la posición index por element.
- *boolean contains(Object o)*; busca el objeto o en la lista y devuelve true si existe. Utiliza el método *equals()* para comparar objetos.
- *int indexOf(Object o)*; busca el objeto o en la lista, empezando por el principio, y devuelve el índice dónde se encuentre. Devuelve -1 si no existe. Utiliza *equals()* para comparar objetos.
- *int lastIndexOf(Object o)*; como *indexOf()* pero busca desde el final de la lista.
- *E remove(int index)*; elimina el elemento en la posición index y lo devuelve.
- *boolean remove(Object obj)*; elimina la primera ocurrencia de obj en la lista. Devuelve true si lo ha encontrado y eliminado, false en otro caso. Utiliza *equals()* para comparar objetos.
- *void remove(int index)*; Elimina el objeto de la lista que se encuentra en la posición index. Es más rápido que el método anterior ya que no necesita recorrer toda la lista.

[Documentación oficial de ArrayList \(Java 11 API\)](#)

1.5 Recorrido de un ArrayList

Hay dos maneras diferentes en las que se puede iterar (recorrer) una colección del tipo *ArrayList*.

- Utilizando el bucle *for* y el método *get()* con un índice.

```
for(int i = 0; i < lista.size(); i++) {  
    System.out.println(lista.get(i)); // Lo imprimimos por pantalla  
}
```
- Usando un objeto *Iterator* que permite recorrer listas como si fuese un índice. Se necesita importar la clase con *import java.util.Iterator*; Tiene dos métodos principales:
 - *hasNext()*: Verifica si hay más elementos.
 - *next()*: devuelve el objeto actual y avanza al siguiente.

```
Iterator iter = lista.iterator(); // Creamos el Iterator a partir de la lista  
while(iter.hasNext()) { // Mientras haya siguiente en la lista  
    System.out.println(iter.next()); // Lo imprimimos por pantalla  
}
```

1.6 Ejemplo 1

Ejemplo que crea, rellena y recorre un *ArrayList* de dos formas diferentes. Cabe destacar que, por defecto, el método *System.out.println()* invoca al método *toString()* de los elementos que se le pasen como argumento, por lo que realmente no es necesario utilizar *toString()* dentro de *println()*.

```
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)
```

1.7 Ejemplo 2

Tenemos la clase Producto con:

- Dos atributos: nombre (String) y cantidad (int).

- Un constructor con parámetros.
- Un constructor sin parámetros.
- 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
```

En el programa principal creamos una lista de productos y realizamos 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)

```

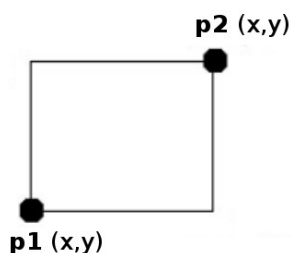
2. COMPOSICIÓN

La composición es el **agrupamiento de uno o varios objetos y valores dentro de una clase**. La composición crea una relación **‘tiene’** o **‘está compuesto por’**.

Por ejemplo, un rectángulo está compuesto por dos puntos (cada uno con sus coordenadas x,y).

```
public class Rectangulo {  
    Punto p1;  
    Punto p2;  
    ...  
}
```

```
public class Punto {  
    int x, y;  
    ...  
}
```



Una cuenta bancaria tiene un titular y un autorizado (ambas son personas con dni, nombre, dirección, teléfono, etc.). Además de el saldo, la cuenta tendrá registrado un listado de movimientos (cada movimiento tiene asociada un tipo, fecha, cantidad, concepto, origen o destino, etc.).

```
public class CuentaBancaria {  
    Persona titular;  
    Persona autorizado;  
    double saldo;  
    Movimiento movimientos[];  
    ...  
}
```

```
public class Persona {  
    String dni, nombre, dirección, teléfono;  
    ...  
}
```

```
public class Movimiento {  
    int tipo;  
    Date fecha;  
    double cantidad;  
    String concepto, origen, destino;  
    ...  
}
```

La composición de clases es una capacidad muy potente de la POO ya que permite diseñar software como un conjunto de clases que colaboran entre sí: Cada clase se especializa en una tarea concreta y esto permite dividir un problema complejo en varios sub-problemas pequeños. También

facilita la modularidad y reutilización del código.

3. HERENCIA

3.1 Introducción

La herencia es una de las capacidades más importantes y distintivas de la POO. Consiste en derivar o **extender una clase nueva a partir de otra ya existente de forma que la clase nueva hereda todos los atributos y métodos de la clase ya existente.**

A la clase ya existente se la denomina **superclase**, clase **base** o clase **padre**. A la nueva clase se la denomina **subclase**, clase **derivada** o clase **hija**.

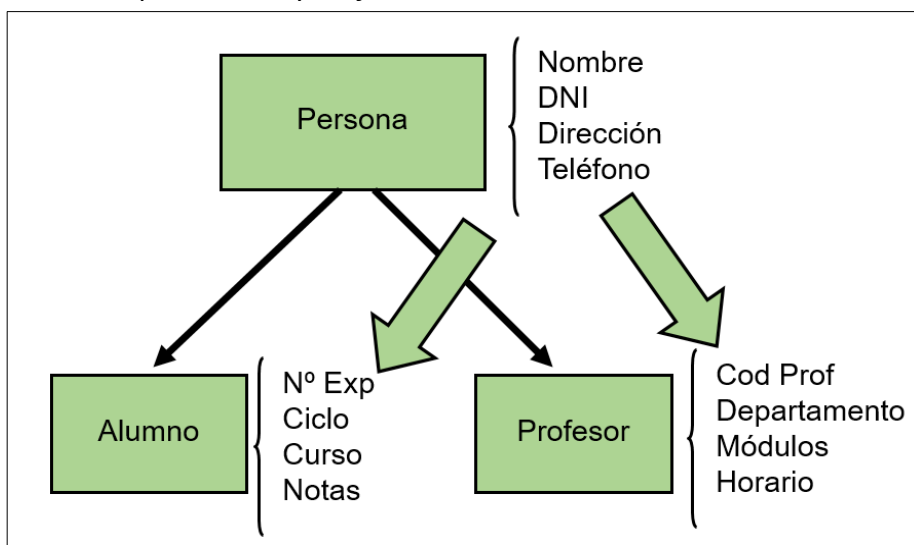


Cuando derivamos (o extendemos) una nueva clase, ésta hereda todos los datos y métodos miembro de la clase existente.

Por ejemplo, si tenemos un programa que va a trabajar con alumnos y profesores, éstos van a tener atributos comunes como el nombre, dni, dirección o teléfono. Pero cada uno de ellos tendrán atributos específicos que no tengan los otros. Por ejemplo 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 declarar una clase *Persona* con los atributos comunes (Nombre, DNI, Dirección, Teléfono) y dos sub-clases *Alumno* y *Profesor* que hereden de *Persona* (además de tener sus propios atributos).

Es importante recalcar que *Alumno* y *Profesor* también heredarán todos los métodos de *Persona*.



En Java se utiliza la palabra reservada **extends** para indicar herencia:

```
public class Alumno extends Persona {  
    ...  
}  
  
public class Profesor extends Persona {  
    ...  
}
```

3.2 Constructores de clases derivadas

El constructor de una clase derivada debe encargarse de construir los atributos que estén definidos en la clase base además de sus propios atributos.

Dentro del constructor de la clase derivada, para llamar al constructor de la clase base se debe utilizar el método reservado ***super()*** pasándole como argumento los parámetros que necesite.

Si no se llama explícitamente al constructor de la clase base mediante ***super()*** el compilador llamará automáticamente al constructor por defecto de la clase base. Si no tiene constructor por defecto el compilador generará un error.

3.3 Métodos heredados y sobreescritos

Hemos visto que una subclase hereda los atributos y métodos de la superclase; además, se pueden incluir nuevos atributos y nuevos métodos.

Por otro lado, puede ocurrir que alguno de los métodos que existen en la superclase no nos sirvan en la subclase (tal y como están programados) y necesitemos adecuarlos a las características de la subclase. Esto puede hacerse mediante la sobreescritura de métodos:

Un método está sobreescrito o reimplementado cuando se programa de nuevo en la clase derivada. Por ejemplo el método *mostrarPersona()* de la clase *Persona* lo necesitaríamos sobrecribir en las clases *Alumno* y *Profesor* para mostrar también los nuevos atributos.

El método sobreescrito en la clase derivada podría reutilizar el método de la clase base, si es necesario, y a continuación imprimir los nuevos atributos. En Java podemos acceder a método definidos en la clase base mediante ***super.metodo()***.

El método *mostrarPersona* sobreescrito en las clases derivadas podría ser:

```
super.mostrarPersona(); // Llamada al método de la clase base
System.out.println(...); // Imprimimos los atributos exclusivos de la clase derivada
```

3.4 Clases y métodos final



Una clase ***final*** no puede ser heredada.



Un método ***final*** no puede ser sobreescrito por las subclases.

3.5 Acceso a miembros derivados

Aunque una subclase incluye todos los miembros de su superclase, no podrá acceder a aquellos que hayan sido declarados como ***private***.

Si en el ejemplo 3 intentásemos acceder desde las clases derivadas a los atributos de la clase *Persona* (que son privados) obtendríamos un error de compilación.

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



Los atributos declarados como ***protected*** son públicos para las clases heredadas y privados para las demás clases.

3.6 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 getters y setters. 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 getters y setters.

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 *Persona*. En la llamada del *main* tanto el objeto *a* (*Alumno*) como el objeto *profe* (*Profesor*) pueden hacer uso del método *mostrarPersona()*.

Persona

```
10 public class Persona {
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 }
```

Alumno

(hereda de Persona)

```
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      public int getExp() {
46          return exp;
47      }
48
49      public void setExp(int exp) {
50          this.exp = exp;
51      }
52
53      public String getCiclo() {
54          return ciclo;
55      }
56
57      public void setCiclo(String ciclo) {
58          this.ciclo = ciclo;
59      }
60
61      public int getCurso() {
62          return curso;
63      }
64
65      public void setCurso(int curso) {
66          this.curso = curso;
67      }
68
69      public ArrayList getNotas() {
70          return notas;
71      }
72
73      public void setNotas(ArrayList notas) {
74          this.notas = notas;
75      }
76  }
```

Profesor (hereda de Persona)

```
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
30      public void mostrarPersona()
31      {
32          // Llamamos al método de la clase madre para que muestre los datos de Persona
33          super.mostrarPersona();
34
35          System.out.println("Código: " + this.cod);
36          System.out.println("Departamento: " + this.depto);
37          System.out.println("Horario: " + this.horario);
38          System.out.println("Modulos:");
39          for( Iterator it = this.modulos.iterator(); it.hasNext(); )
40          {
41              System.out.println("\tMódulo: " + it.next());
42          }
43      }
44
45      public int getCod() {
46          return cod;
47      }
48
49      public void setCod(int cod) {
50          this.cod = cod;
51      }
52
53      public String getDepto() {
54          return depto;
55      }
56
57      public void setDepto(String depto) {
58          this.depto = depto;
59      }
60
61      public ArrayList getModulos() {
62          return modulos;
63      }
64
65      public void setModulos(ArrayList modulos) {
66          this.modulos = modulos;
67      }
68
69      public String getHorario() {
70          return horario;
71      }
72
73      public void setHorario(String horario) {
74          this.horario = horario;
75      }
76  }
```

Programa Principal

```

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("Maria", "123456782", "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:

```

Output - Herencia (run) X
run:
Mostramos una persona
Nombre: Pepe
DNI: 00000000T
Dirección: C/ Colón
Teléfono: 666666666
-----
Mostramos un alumno
Nombre: Maria
DNI: 123456782
Dirección: P/ Libertad
Teléfono: 11111111
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: 22222222
Código: 3
Departamento: Informática
Horario: Mañanas
Modulos:
    Módulo: Programación
    Módulo: Lenguajes de marcas
    Módulo: Entornos de desarrollo
BUILD SUCCESSFUL (total time: 0 seconds)

```

4. 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 en tiempo de ejecución y no durante la 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 fundamental 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

4.1 Ejemplo 4

Vamos probar un ejemplo sencillo pero que resume todo lo importante del polimorfismo.

Vamos a crear la **clase Madre** con un método *llamame()*. A continuación crearemos **dos clases derivadas** de ésta: **Hija1** e **Hija2**, sobreescribiendo el método *llamame()*. En el *main* crearemos un objeto de cada clase y los asignaremos a una variable de tipo *Madre* (llamada *madre2*) con la que llamaremos al método *llamame()* de los tres objetos.

Es importante observar que la variable Madre madre2 se puede asignar a objetos de clase Hija1 e Hija2. Esto es posible porque Hija1 e Hija2 también son de tipo Madre (debido a la herencia).

También es importante ver que la variable Madre madre2 llamará al método llamame() de la clase del objeto al que hace referencia (debido al polimorfismo).

Obsérvese las llamadas *madre2.llamame()* de las líneas 36 en adelante:

- En el primero se invoca al método *llamame()* de la clase Madre porque 'madre2' hace referencia a un objeto de la clase Madre.
- En el segundo se invoca al método *llamame()* de la clase Hija1 porque ahora 'madre2' hace referencia a un objeto de la clase Hija1.
- En el tercero se invoca al método *llamame()* de la clase Hija2 porque ahora 'madre2' hace referencia a un objeto 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)
```

5. CLASES ABSTRACTAS

Una clase abstracta es **una clase que declara la existencia de algunos métodos pero no su implementación** (es decir, contiene la cabecera del método pero no su código). Los métodos sin implementar son métodos abstractos.

Una clase abstracta puede contener tanto métodos abstractos (sin implementar) como no abstractos (implementados). Pero al menos uno debe ser abstracto.

Para declarar una clase o método como abstracto se utiliza el modificador **abstract**.

⚡ Una clase abstracta **no se puede instanciar**, pero **sí heredar**. Las subclases tendrán que implementar obligatoriamente el código de los métodos abstractos (a no ser que también se declaren como abstractas).

Las clases abstractas son útiles cuando necesitemos definir una forma generalizada de clase que será compartida por las subclases, dejando parte del código en la clase abstracta (métodos “normales”) y delegando otra parte en las subclases (métodos abstractos).

⚡ No pueden declararse constructores o métodos estáticos abstractos.

La finalidad principal de una clase abstracta es crear una clase heredada a partir de ella. Por ello, en la práctica es obligatorio aplicar herencia (si no, la clase abstracta no sirve para nada). El caso contrario es una clase *final*, que no puede heredarse como ya hemos visto. Por lo tanto una clase no puede ser *abstract* y *final* al mismo tiempo.

Por ejemplo, esta clase abstracta Principal tienes dos métodos: uno concreto y otro abstracto.

```
public abstract class Principal {  
    // Método concreto con implementación  
    public void metodoConcreto() {  
        ...  
    }  
    // Método abstracto sin implementación  
    public abstract void metodoAbstracto();  
}
```

Esta subclase hereda de Principal ambos métodos, pero está obligada a implementar el código del método abstracto.

```
class Secundaria extends Principal {  
    // Implementación concreta  
    public void metodoAbstracto() {  
        ...  
    }  
}
```

6. INTERFACES

Una interfaz es una **declaración de atributos y métodos sin implementación** (sin definir el código de los métodos). Se utilizan para definir el conjunto mínimo de atributos y métodos de las clases que implementen dicha interfaz. En cierto modo, es parecido a una clase abstracta con todos sus miembros abstractos.

Si una clase es una plantilla para crear objetos, **una interfaz es una plantilla para crear clases.**



Un interfaz es una declaración de atributos y métodos sin implementación.

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

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

La sintaxis típica de una interfaz es la siguiente:

```
public interface Nombre {  
    // Declaración de atributos y métodos (sin definir código)  
}
```

Si una interfaz 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:

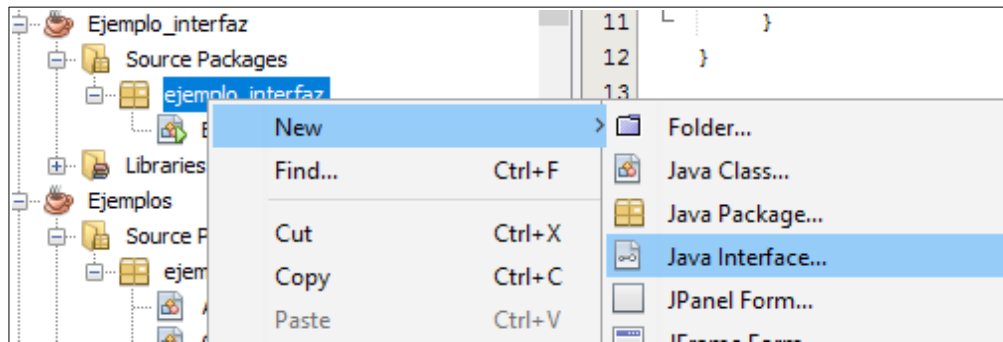
- Una interfaz no se puede instanciar en objetos, solo sirve para implementar clases.
- Una clase puede implementar varias interfaces (separadas por comas).
- Una clase que implementa una interfaz debe de proporcionar implementación para todos y cada uno de los métodos definidos en la interfaz.
- Las 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 *Círculo* implementa la interfaz *Figura* la sintaxis sería:

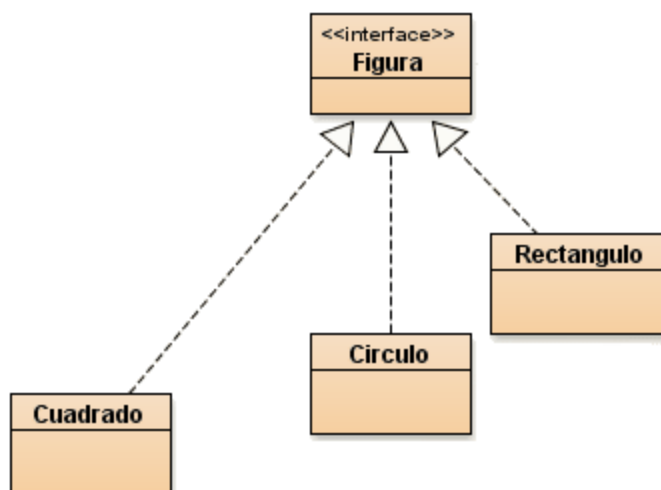
```
public class Círculo implements Figura {  
    ...  
}
```

6.1 Ejemplo 6

En este ejemplo vamos a crear una interfaz *Figura* y posteriormente implementarla en varias clases. 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  
}
```

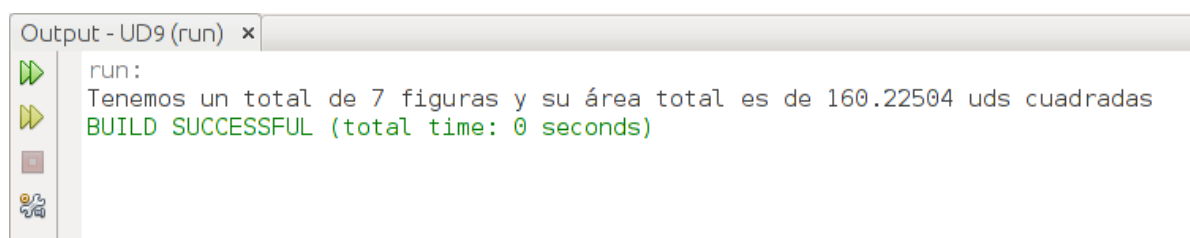
```
12 public class Cuadrado implements Figura {  
13     private float lado;  
14  
15     public Cuadrado (float lado) {  
16         this.lado = lado;  
17     }  
18  
19     public float area() {  
20         return lado*lado;  
21     }  
22 }  
23
```

```
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 **la interface Figura define un tipo de dato**. Por ello podemos crear un ArrayList de figuras donde insertamos cuadrados, círculos, rectángulos, etc. (polimorfismo). Esto nos permite darle un tratamiento común a todas las figuras: Mediante un bucle while recorreremos la lista de figuras y llamamos al método area() que será distinto para cada clase de figura.