6. TEMA 6: Mecanismos de abstracción. Clases, paquetes, subclases e interfaces.

En este tema aprenderemos a manejar los diferentes métodos de abstracción que nos proporciona Java. Entre los cuales encontramos la herencia y el polimorfismo.

En cuanto a la herencia veremos sus ventajas y como afecta esto a la reutilización de código.

Usaremos clases heredadas y crearemos las nuestras propias. Veremos como forzar el uso de la herencia y como se realiza la herencia múltiple.

Debemos identificar también cuando es necesaria la herencia o el polimorfismo en cada caso.

Identificaremos y usaremos clases abstractas e interfaces.

Antes de comenzar con este tema se va a hacer un pequeño inciso sobre la forma en la que se van a presentar los ejemplos del libro a partir de ahora. Para que el código de ejemplo no sea complejo y poder ir a las partes concretas e interesantes se van a omitir los métodos “get” y “set” de los distintos atributos privados en ciertos ejemplos. Además también se va a crear el método main en la clase que se crea apropiada para su ejecución, de tal manera que no es necesario crear un archivo aparte para el método “main”. De tal manera que nos encontraremos casos de ejemplo como este:

```
public class Mascota {
	private String nombre;

	public Mascota(String nom){
		nombre=nom;
	}
	public void mostrar(){
		System.out.println("Mascota:"+nombre);
	}

	// Aquí estarían los métodos “get” y “set” de los atributos privados

	public static void main(String[] args) {

		Mascota a1=new Alumno("Noa");
		a1.mostrar();
	}
}
```

Como se puede observar en una misma clase estamos usando el método “main” para crear una instancia de esa misma clase. Esto puede empezar a usarse a partir de ahora, al igual deberá suponerse que existen los métodos “get” y “set” tal y como se especifica en los comentarios.

6.1 Relaciones entre clases.

Antes de empezar con la reutilización y la herencia debemos conocer como se relacionan las clases entre ellas.

Relación entre clases → Esta es la relación en la que tiene lugar la herencia. Imaginemos que tenemos una clase creada la cual consta de una serie de atributos y métodos, y posteriormente necesitamos otra clase que tiene las mismas características pero que necesita tener un atributo o un método más, para esto usaremos la herencia. En definitiva usaremos la herencia cuando deseamos ampliar la funcionalidad de una clase concreta y partimos de ella aprovechando lo que ya contiene.

Veamos esto con un ejemplo más concreto. Supongamos que tenemos la clase "Persona" que tiene un nombre y una edad y queremos crear una clase "Empleado" que además de lo que tiene "Persona" tiene un sueldo. Veamos como sería esto:

La clase Persona.java:

```
public class Persona {
	private String nombre;
	private int edad;
	public Persona(String n,int e){
		nombre=n;
		edad=e;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}

	public void mostrar(){
		System.out.println("Nombre:"+nombre+"\tedad:"+edad);
	}
}
```

La clase Empleado.java:

```
public class Empleado extends Persona{
	private float sueldo;
	public Empleado(String nombre,int edad,float sueldo){
		super(nombre,edad);
		this.setSueldo(sueldo);
	}
	public float getSueldo() {
		return sueldo;
	}
	public void setSueldo(float sueldo) {
		this.sueldo = sueldo;
	}
}
```

Como puede observarse hay una palabra que hasta ahora nos era desconocida, se trata de "**extends**", esta palabra reservada del lenguaje indica que ahora "Empleado"

hereda de “Persona”, por tanto hereda todos sus atributos y métodos. Otra palabra que nos resultará extraña es “**super**”, esta palabra se usa para hacer referencia a la clase de la cual heredamos, en este caso “Persona”, en ese ejemplo concreto lo que hacemos es pasar los parámetros “nombre” y “edad” al constructor de “Persona”.

Relación entre instancias → Consiste en usar instancias de otros objetos para intercambiar información o completar un objeto si lo que estamos haciendo es una composición. Este tipo de relaciones no tienen nada que ver con la herencia. En función de como los organicemos se tratará de un tipo de relación u otro. Veamos las posibles relaciones con las que nos encontraremos:

- **Relación de dependencia** → Imaginemos que tenemos una clase “Cuenta”, y esta tiene un método “sacarDinero” el cual nos pide que por teclado introduzcamos la cantidad, por tanto para poder hacer esto tenemos que usar la clase “Scanner”. Por tanto podemos decir que la clase “Cuenta” depende de la clase “Scanner”. Veámoslo con un ejemplo:

Tenemos la clase “Cuenta.java”, donde tenemos un “main” de ejemplo:

```
import java.util.Scanner;

public class Cuenta {
	private String dniCliente;
	private float saldo;
	private long numCuenta;

	public Cuenta(String cliente,float saldo,long num){
		dniCliente=cliente;
		this.saldo=saldo;
		numCuenta=num;
	}

	public void sacarDinero(){
		Scanner sc=new Scanner(System.in);
		float cantidad;
		System.out.println("Tu saldo actual es:"+saldo+" euros");
		do{
			System.out.print("Cuanto dinero quieres retirar de la
cuenta?:");
			cantidad=sc.nextFloat();
			if(cantidad>saldo){
				System.err.println("No tienes suficiente saldo.");
			}
		}while(cantidad>saldo);
		saldo=saldo-cantidad;
		System.out.println("Ahora tu saldo actual es:"+saldo+"
euros");
	}
	public static void main(String[] args){
		Cuenta miCuenta=new Cuenta("47543674F",578.7f,236532672);
		miCuenta.sacarDinero();
	}
}
```

Lo que podemos observar es que para ejecutar el método sacar dinero se crea una

cierta dependencia de la clase “Cuenta” hacia la instancia de la clase “Scanner”.

- **Relación de asociación** → Dos objetos que están relacionados entre si, pero son independientes el uno del otro. Vamos a imaginar que realizamos un programa que gestiona un colegio. En dicho programa tenemos una clase “Alumno” y otra “Profesor”. La clase “Alumno” puede tener un atributo que sea de tipo “Profesor”, que sería el profesor que este alumno tiene asignado. Pero el echo de que el alumno se dé de baja no quiere decir que el profesor tenga que desaparecer. Además este tipo de relación puede ser unidireccional como bidireccional.

  Tenemos la clase “Profesor.java”:

```
public class Profesor {
	private String nombre;
	private String titulo;
	private String asignatura;

	public Profesor(String nom,String tit,String asig){
		nombre=nom;
		titulo=tit;
		asignatura=asig;
	}

	public void mostrar(){
		System.out.println("PROFESOR->Nombre:"+nombre+"\tTitulo:"+
				titulo+"\tAsignatura:"+asignatura);
	}
}
```

  Y la clase “Alumno.java”, donde tenemos el “main” de ejemplo:

```
public class Alumno {
	private String nombre;
	private Profesor p;

	public Alumno(String nom,Profesor prof){
		nombre=nom;
		p=prof;
	}
	public void mostrar(){
		System.out.println("ALUMNO:"+nombre+
			"\n*****************************");
		p.mostrar();
	}
	public static void main(String[] args) {
		Profesor p1=new Profesor("Pepe","Ingeniero
industrial","Mecanica");
		Profesor p2=new Profesor("Juan","Ingeniero
Químico","Química");
		Profesor p3=new Profesor("Jose","Licenciado en
derecho","FOL");
		Alumno a1=new Alumno("Antonio",p1);
		Alumno a2=new Alumno("Marcos",p3);
		Alumno a3=new Alumno("Maria",p1);
```

```
	}
}
```

Como se puede observar, para crear un "Alumno" debemos asociarlo a un "Profesor", pero el hecho de que no exista ningún alumno que tenga asignado al profesor, como es el caso del profesor "Juan" en el ejemplo, esto no implica que esa clase no exista. Hay una relación de asociación pero el profesor "Juan" es independiente de que existan o no alumnos asignados a su clase. Este tipo de relación sería unidireccional, en el caso de que el profesor tuviese un conjunto de alumnos asignados y hubiese una referencia a estos alumnos en la clase "Profesor" esto implicaría que la relación es bidireccional, pero que un alumno no esté asignado a ese profesor no implica tampoco que ese alumno no pueda existir, dado que podría estar asignado a cualquier otro profesor.

- **Relación todo-parte** → En este caso lo podemos dividir en dos subtipos de relaciones:
  - **Agregación**: Es un tipo especial de asociación en la cual tenemos un objeto que es un "todo" y otro que es una "parte". La desaparición del "todo" no implica la desaparición de la "parte" que puede estar asociada a otras partes. Por ejemplo vamos a imaginar que tenemos un programa que gestiona un banco, y la condición indispensable es que para dar de alta a un "Cliente" debe tener al menos una "Cuenta". Por tanto tenemos estos dos objetos, y el objeto "Cliente" va a estar siempre asociado a un objeto "Cuenta". Si eliminamos una cuenta no implica que tenga que eliminarse el cliente, en cambio si ya no existen más cuentas asociadas a ese cliente debería eliminarse el cliente.

    De nuevo intentaremos ilustrar este ejemplo mediante código. Suponemos que tenemos la clase cuenta mencionada en los apartados anteriores y además tenemos la clase "Cliente.java":

```
import java.util.ArrayList;

public class Cliente {
	private String dni;
	private ArrayList<Cuenta> cuentas;

	public Cliente(String dni,float dinero){
		this.dni=dni;
		cuentas=new ArrayList<Cuenta>();
		Cuenta c=new Cuenta(dni,dinero,(long)(Math.random()*9999999));
		cuentas.add(c);
	}

	public void agregarCuenta(float dinero){
		Cuenta c=new Cuenta(this.dni,dinero,(long)
(Math.random()*9999999));
		cuentas.add(c);
	}

	public static void main(String[] args) {
		Cliente c1=new Cliente("36752843D",500);
		c1.agregarCuenta(250);
	}
}
```

Como podemos observar se crea un cliente, al cual se le asigna una cuenta, y a continuación se le agrega otra cuenta. El cliente para ser creado necesita tener al menos una cuenta, después vamos agregando diferentes cuentas, que las cuales son las "partes" que forman el "todo", en el momento en que todas las partes son eliminadas el "todo" deja de existir, por tanto el cliente ya no debería existir.

- **Composición**: Aunque conceptualmente es parecido a todo lo anterior, la composición se diferencia en que dentro de una clase definimos otra clase, de tal manera que si la clase contenedor desaparece también desaparece la clase contenida.

```
public class Composicion {
	private String nombre;
	private Apellido ap1;
	private Apellido ap2;

	// ESTO ES LA COMPOSICIÓN
	class Apellido{
		private String ape;
		public Apellido(String ape){
			this.ape=ape;
		}
	}

	public Composicion(String nombre,String ape1,String ape2){
		this.nombre=nombre;
		ap1=new Apellido(ape1);
		ap2=new Apellido(ape2);

	}
	public static void main(String[] args) {
		Composicion c=new Composicion("Andres","Terol","Sanchez");
	}
}
```

Como se observa dentro de la clase "Composicion" tenemos otra clase creada llamada "Apellido", estas serían las "partes" que forman el "todo" en "Composicion". La clase "Apellido" solo tiene sentido dentro de la clase "Composicion".

6.2 Herencia.

Como se ha dicho anteriormente la herencia vamos a usarla cuando necesitamos ampliar la funcionalidad de una clase creada anteriormente. Podemos acceder a los atributos y métodos de la clase heredada dependiendo de la visibilidad que tengan estos.

Por tanto la herencia nos permite:

- La reutilización de código.
- Añadir nuevos datos y métodos a la clase.
- Modificar el método de algún método de la clase heredada, mediante la sobre escritura del método.

- Generar una jerarquía de clases.

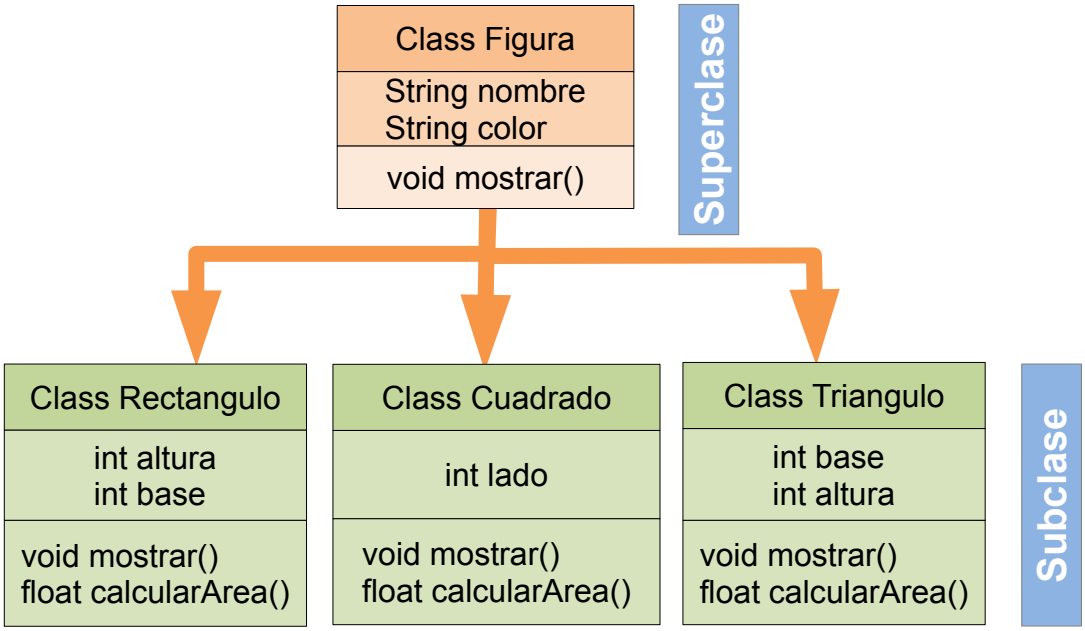
Existen dos tipos de herencia:

- La herencia **simple**, implica que solo heredaremos de una clase.
- La herencia **múltiple**, implica que heredaremos de varias clases.

Java sólo permite la herencia simple, **no permite herencia múltiple,** pero a cambio dispone de la construcción denominada "Interface" que permite una forma de simulación o implementación limitada de la herencia múltiple.

Básicamente la herencia responde a la relación de tipo "**es-un**", por ejemplo un Alumno ***es-una*** Persona.

Generalmente a la clase de la cual heredamos la llamaremos "superclase" y la clase heredada la llamaremos "subclase". Vamos a ilustrarlo con el siguiente esquema:



Como vemos el esquema anterior responde a la relación Rectangulo ***es-una*** Figura.

La implementación para el esquema visto arriba sería el siguiente, primero creamos la clase "Figura.java" en la cual los atributos van a ser "protected":

```
public class Figura {
	protected String nombre,color;

	public Figura(String nom,String col)
	{
		nombre=nom;
		color=col;
	}

	public void mostrar()
	{
		System.out.println("Soy la Figura:"+nombre+" de color "+color);
	}
}
```

Hasta aquí todo normal, exceptuando el "protected", que en el siguiente apartado veremos mas detenidamente para qué sirve.

Ahora vamos a crear nuestra primera subclase de la superclase "Figura", en concreto la clase "Rectangulo.java":

```
public class Rectangulo extends Figura{
	protected float base, altura;

	public Rectangulo(String nombre, String color,float b, float a)
	{
		super(nombre, color);
		this.base=b;
		this.altura=a;
	}

	public void mostrar()
	{
		System.out.println("Soy el rectángulo:"+nombre+" de color "+color
				+" y mi área es:"+calcular_area());
	}

	public float calcular_area()
	{
		return (base*altura);
	}
}
```

De aquí cabe destacar lo siguiente, primero debemos observar que cuando definimos la subclase lleva la palabra "**extends**" y a continuación el nombre de la superclase, la clase de la cual hereda. A continuación podemos ver como el constructor tiene una cierta particularidad, y es que aunque tiene como parámetros base y altura, también le pasamos el nombre y el color, esto es porque esos dos atributos al heredarlos de la superclase "Figura" también tendrá que inicializarlos en el constructor, lo peculiar del caso es la forma de inicializarlos, como vemos se hace mediante "**super**(nombre, color)", cuando usamos la palabra reservada "**super**" dentro de una subclase lo que estamos haciendo es llamar a la superclase, por tanto en este caso lo que hacemos es pasarle los atributos "nombre" y "color" al constructor de la superclase "Figura" de tal manera que se encarga ese constructor de inicializarlos. Un apunte importante es que cuando creamos el constructor de la subclase la primera llamada o instrucción debe ser al constructor de la superclase como se observa en el código arriba expuesto.

Como último dato importante tenemos el método "mostrar", si recordamos, en "Figura" también teníamos un método mostrar, entonces, ¿que sucede?, lo que hemos hecho es sobrescribir el método "mostrar" de la superclase "Figura", de tal manera que cuando un objeto de tipo "Rectangulo" llama al método "mostrar" está llamando al método de "Rectangulo", no al de "Figura". También podemos observar como podemos usar sin problemas los atributos "nombre" y "color" heredados de la superclase "Figura".

Por último podríamos preguntarnos, ¿que sucede si quiero sobrescribir un método y además quiero reutilizar el código que tenía el método de la superclase?. La respuesta es sencilla, bastaría con hacer una llamada al método "mostrar" de "Figura" a través de la palabra reservada "super", tal y como lo hacemos en el siguiente ejemplo en el que el método mostrar nos quedaría de la siguiente manera:

```
public void mostrar()
{
	super.mostrar();
	System.out.println("Soy el rectángulo:"+nombre+" de color "+color
			+" y mi área es:"+calcular_area());
}
```

Ejercicio Auto-evaluación A6.1: *Realiza las clases "Cuadrado.java" y "Triangulo.java" para completar el ejemplo de las figuras.*

Otra cualidad que tiene la herencia es que yo puedo instanciar cualquiera de las subclases dentro de un objeto de la superclase, por tanto puedo hacer cosas como estas:

```
public static void main(String[] args){
	Figura rect=new Rectangulo("Rect1","Verde",20,30);
	rect.mostrar();
}
```

Curiosamente puedo introducir la instancia del "Rectangulo" dentro del objeto "rect" que es de "Figura", y aún mas, el método mostrar funciona correctamente. Esto es debido al concepto de ligadura dinámica que abordaremos más adelante, pero básicamente quiere decir que en tiempo de ejecución en función del tipo de instancia de la superclase nos liga a los métodos de la subclase instanciada, de tal manera que si en una figura meto un rectángulo y llamo a su mostrar se ejecutará el método correcto siempre y cuando ese método exista en "Figura", e incluso desde el mostrar se llamará al método "calcular_area" y se invocará correctamente. Lo que no se podrá hacer es lo siguiente:

```
public static void main(String[] args){
	Figura rect=new Rectangulo("Rect1","Verde",20,30);
	rect.mostrar();
	rect.calcular_area();
}
```

Esto nos provocará un error debido a que no se encuentra el método "calcular_area", debido a que en "Figura" no existe tal método y no se puede hacer por tanto ningún tipo de ligadura.

Otra cosa que tampoco podremos hacer es la siguiente:

```
public static void main(String[] args){
	Rectangulo rect=new Figura("Rect1","Verde");
	rect.mostrar();
}
```

Por lógica no puedo guardar una "Figura" en un "Rectangulo" ya que "Rectangulo" tiene dos atributos más que "Figura" y estos se quedarían sin instanciarse.

6.2.1 Visibilidad de los atributos en herencia.

En el "Tema 4" vimos una tabla en la cual teníamos la visibilidad de los atributos, y se hizo caso omiso a aquellos que estaban marcados en naranja debido a que para entenderlos antes debíamos abordar el tema de la herencia:

Método o Atributo	public	protected	private	Sin especificar (package)
¿Accesible desde la propia clase?	Si	Si	Si	Si
¿Accesible desde otras clases en el mismo paquete?	Si	Si	No	Si
¿Accesible desde una subclase en el mismo paquete?	Si	Si	No	Si
¿Accesible desde subclases en otros paquetes?	Si	(*)	No	No
¿Accesible desde otras clases en otros paquetes?	Si	No	No	No

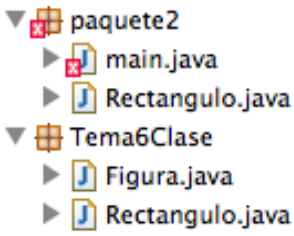
(*) Se puede acceder por los objetos de la subclase pero no por los de la superclase.

Dada la tabla anterior lo que podemos observar es que con atributos o métodos "protected" tenemos accesibilidad o no dependiendo si las subclases están en el mismo paquete o no, al igual sucede que sino especificamos la visibilidad nunca tendremos visibilidad desde subclases de otros paquetes.

Veamos un ejemplo, vamos a suponer que tenemos la clase "Figura.java" y "Rectangulo.java" en un paquete llamado "Tema6Clase" y a continuación creo otro paquete diferente llamado "paquete2" en el cual creo de nuevo una clase "Rectangulo.java" que hereda de "Figura.java" del paquete "Tema6Clase" y todos tienen sus atributos como protected, y a parte tengo el siguiente "main.java":

```
public class main {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Figura f1=new Figura("nuevo","azul");
		Rectangulo r1=new Rectangulo("rect","amarillo",20,40);
		r1.altura=5;
		f1.color="verde";
	}
}
```

La jerarquía de paquetes es esta:



Como se observa el "main.java" tiene un error, y es porque aunque puedo acceder a los atributos propios del "Rectangulo" pero no a los que pertenecen a "Figura" que está en otro paquete diferente.

Respecto a la visibilidad de métodos en la herencia cabe decir que una subclase puede redefinir cualquiera de los métodos de la superclase siempre que no sean "**final**". El nuevo método sustituye al heredado en todos los aspectos. Los métodos redefinidos pueden ampliar los derechos de acceso o visibilidad de la superclase, por ejemplo ser public en vez de protected o private, pero nunca pueden restringirlos.

Los métodos de clase o static nunca pueden ser redefinidos.

Todo lo anterior podríamos sintetizarlo en la siguiente tabla:

Visibilidad en Superclase	Posible visibilidad en Subclase
private	private package protected public
package	package protected public
protected	protected public
public	public

Ejercicio Auto-evaluación A6.2: *Crear una clase Vehiculo que tiene los atributos color y marca que son cadenas de texto, y tenemos un mostrar que nos muestra los atributos de Vehiculo. A continuación crear 3 subclases:*

Coche*: que tiene como atributo numRuedas de tipo entero y sobrescribe el método mostrar, además del "get" y "set" para numRuedas.*

Barco*: que tiene como atributo eslora de tipo float y sobrescribe el método mostrar, además del "get" y "set" para eslora.*

Avion*: que tiene como atributo helices de tipo entero y sobrescribe el método mostrar, además del "get" y "set" para helices.*

6.3 Forzando la herencia. Clases abstractas.

Tal y como se ha visto la herencia nos aporta muchas ventajas diferentes, pero otra que nos ofrece es la de poder forzar la sobrescritura de ciertos métodos de la superclase para poder heredar de ella, de tal manera que podemos forzar que todas nuestras subclases implementen una interfaz básica común.

Lo primero que debemos tener en cuenta es que cualquier clase que tenga métodos abstractos debe definirse como una clase abstracta. **Una clase abstracta nunca puede instanciarse.**

Los métodos abstractos no tienen implementación dentro de la superclase, solo están definidos, se implementan en las subclases.

Veamos entonces un ejemplo de clase abstracta, siguiendo con el ejemplo expuesto en el ejercicio de "*Auto-evaluación 6.2*", ahora queremos crear dos coches que heredan de coche, uno va a ser un coche de gasolina y el otro un coche eléctrico. El coche de gasolina va a tener un atributo de tipo decimal llamado deposito y el coche eléctrico va a tener un atributo de tipo entero llamado batería. Ambos coches van a tener un método

llamado correr, en el caso del coche de gasolina cada vez que corramos el depósito se va a decrementar en 10, y en el caso del eléctrico la batería se decrementará en 2. Por tanto como vemos ambos coches tienen que mantener una interfaz común que sería el método "correr", así que lo lógico es que este método esté como abstracto en la clase "Coche", por tanto tenemos la clase "Coche.java":

```
public abstract class Coche extends Vehiculo {
	private int numRuedas;

	public Coche(String m,String c,int r){
		super(m,c);
		numRuedas=r;
	}

	public void setNumRuedas(int r){
		numRuedas=r;
	}

	public int getNumRuedas(){
		return numRuedas;
	}

	public void mostrar(){
		System.out.println("COCHE\n***************");
		super.mostrar();
		System.out.println("Num.Ruedas:"+numRuedas);
	}

	public abstract void correr();
}
```

Como vemos la clase está puesta como "abstract", y además también indicamos en el método "correr" que es "abstract", a parte que podemos observar que no tiene implementación alguna, ya que esta deberán hacerlas las subclases. A partir de este momento no puedo crear ninguna instancia de coche, sólo puedo de las subclases.

Vamos con el resto, el "CocheElectrico.java":

```
public class CocheElectrico extends Coche{
	private int bateria;

	public CocheElectrico(String marca,String color,int numRuedas,int bateria)
{
		super(marca,color,numRuedas);
		this.bateria=bateria;
	}

	public void setBateria(int bateria){
		this.bateria=bateria;
	}
	public int getBateria(){
		return bateria;
	}
	@Override
	public void correr(){
```

```
            if (bateria<2){
                  System.out.println("No hay bateria suficiente");
            }else{
                  System.out.println("El coche eléctrico corre.");
                  this.mostrar();
                  bateria-=2;
            }
      }

      public void mostrar(){
            super.mostrar();
            System.out.println("Tienes "+bateria+" de bateria");
      }
}
```

Podemos observar que el método correr lleva arriba un texto que pone "***@Override***", esto indica que estamos sobrescribiendo el método.

Ahora veamos el "CocheGasolina.java":

```
public class CocheGasolina extends Coche{
      private float deposito;

      public CocheGasolina(String marca,String color,int numRuedas,float
deposito){
            super(marca,color,numRuedas);
            this.deposito=deposito;
      }

      public void setDeposito(float deposito){
            this.deposito=deposito;
      }
      public float getDeposito(){
            return deposito;
      }
      @Override
      public void correr(){
            if (deposito<10){
                  System.out.println("No hay combustible suficiente");
            }else{
                  System.out.println("El coche de gasolina corre.");
                  this.mostrar();
                  deposito-=10;
            }
      }

      public void mostrar(){
            super.mostrar();
            System.out.println("Tienes "+deposito+" litros de combustible");
      }
}
```

Y ahora un ejemplo del main:

```
      public static void main(String[] args) {
            CocheElectrico ce=new CocheElectrico("VW","Rojo",3,7);
```

```
        CocheGasolina cg=new CocheGasolina("Volvo","Gris",4,35);

        ce.correr();
        ce.correr();
        ce.correr();
        ce.correr();
        ce.mostrar();
        cg.correr();
        cg.correr();
        cg.correr();
        cg.correr();
        cg.mostrar();
    }
```

Ejercicio Auto-evaluación A6.3: *Teniendo en cuenta que ya tenemos una clase Persona que tiene un nombre y una edad, realizar una clase Empleado que va a tener lo mismo que persona y a parte un identificador de empleado y un sueldo. Además tendremos dos tipos de empleados en nuestra empresa, unos serán secretarios y otros comerciales.*

Los secretarios se les calcula el sueldo en función de las horas, cada hora cobran 7 euros.

Los comerciales se les cobra el sueldo en función de las horas también, cada hora cobran 6 euros, más a parte los desplazamientos, cada kilómetro en coche lo cobran a 0'30 céntimos de euro.

Realizar un ejemplo y visualizar el sueldo de un secretario que ha echado 160 horas, y un comercial que ha echado 150 horas y 2.000 kilómetros.

6.4 Interfaces.

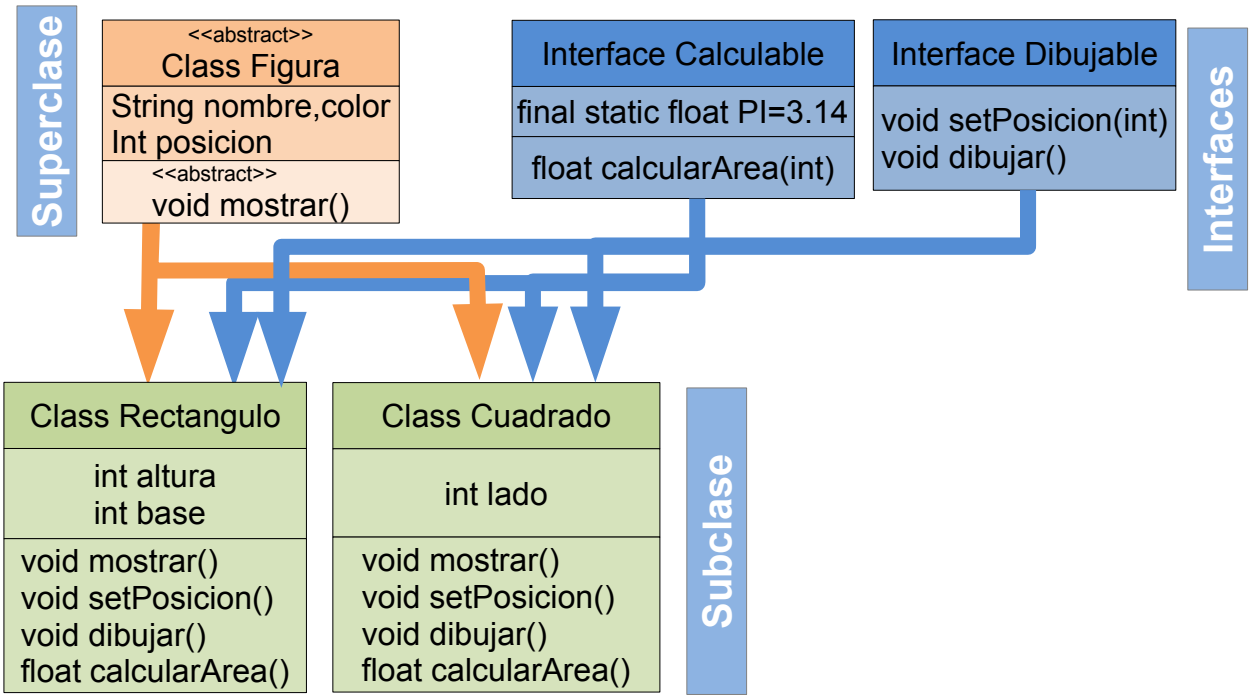
Una interface es un conjunto de declaraciones de métodos sin implementación. También pueden definir constantes que son implícitamente public, static y final. Las interfaces lo que hacen es definir un tipo de conducta, de tal manera que aquella clase que implementa una interfaz está obligada a implementar el comportamiento de la misma y de estos métodos. Por tanto cuando una clase use una interfaz tendrá que sobrecargar sus métodos con acceso público.

En estos momentos podemos pensar que lo que nos ofrecen las interfaces es muy similar a lo que hemos visto en el apartado anterior con abstract, entonces, ¿qué diferencia hay entre una interface y una clase abstract?. Ambas tienen en común que pueden definir un comportamiento, con la diferencia de que en abstract nos permite implementar incluso algunos de los métodos. A pesar de todo esto existen algunas diferencias importantes:

1. Las interfaces nos permiten la herencia múltiple. Una clase puede implementar varias interfaces o una clase abstract y varias interfaces, en cambio una clase no puede heredar de varias clases abstractas.
2. Una clase no puede heredar atributos de una interfaz, aunque si puede heredar constantes.

3. Las interfaces nos ofrecen más flexibilidad, tienen una jerarquía a parte, propia e independiente de las clases.
4. Las interfaces nos permiten publicar el comportamiento con un mínimo de información.
5. De cara al siguiente apartado de polimorfismo las interfaces se pueden usar de forma similar a las clases abstractas.

Vamos a ver como se define una interfaz. Para ello nos valdremos de uno de los ejemplos que hemos estado usando en este tema como es el de las Figuras. La idea será implementar el siguiente esquema:



De lo que se trata es de intentar poner en práctica un poco todo lo visto hasta ahora agregando el concepto de interfaces. Esta vez el método mostrar de “Figura” va a a ser abstracto de tal manera que tendrán que implementarlo tanto “Rectangulo” como “Cuadrado”. Lo mas innovador en este caso serán las interfaces “Dibujable” y “Calculable” que nos van a proporcionar el marco común para “Rectangulo” y “Cuadrado”.

Así nos quedaría “Figura.java”:

```
public abstract class Figura {
	protected String nombre,color;
	protected int posicion;
	public Figura(String nom,String col, int pos)
	{
		nombre=nom;
		color=col;
		posicion=pos;
	}

	public abstract void mostrar();
}
```

Así quedaría la interfaz “Dibujable.java”:

```
public interface Dibujable {
	public void setPosicion(int pos);
	public void dibujar();
}
```

Así quedaría la interfaz “Calculable.java”:

```
public interface Calculable {
	float PI=3.14f;
	public float calcularArea();
}
```

Así quedaría la clase “Cuadrado.java”:

```
public class Cuadrado extends Figura implements Dibujable,Calculable{
	protected float lado;

	public Cuadrado(String nombre, String color,int posicion,float l)
	{
		super(nombre, color,posicion);
		this.lado=l;
	}
	@Override
	public void mostrar()
	{
		System.out.println("Soy el cuadrado:"+nombre+" de color "+color
				+" y mi área es:"+calcularArea());
	}

	@Override
	public float calcularArea() {
		return (lado*lado);
	}
	@Override
	public void setPosicion(int pos) {
		posicion=pos;
	}
	@Override
	public void dibujar() {
		String desplaza="";
		for(int i=0;i<posicion;i++){
			desplaza=desplaza+"  ";
		}
		for(int i=0;i<lado;i++){
			System.out.print(desplaza);
			for(int j=0;j<lado;j++){
				System.out.print("* ");
			}
			System.out.println();
		}
	}

	public static void main(String[] args){
		Cuadrado c1=new Cuadrado("c1","Verde",1,10);
```

```
            c1.mostrar();
            c1.dibujar();
            c1.setPosicion(10);
            c1.dibujar();
        }
}
```

Como se puede observar heredamos de "Figura" e implementamos su método abstracto "mostrar", pero además también implementamos las interfaces "Dibujable" para dibujar la figura con o sin desplazamiento en la posición y "Calculable" para calcular el área.

Ejercicio Auto-evaluación A6.4: *Realiza la clase "Rectangulo.java" para completar el ejemplo de las figuras.*

6.5 Polimorfismo.

La etimología de la palabra "polimorfismo" proviene de "poli" que significa muchas, y "morfo" que significa forma, por tanto podemos deducir que el término hace referencia a la capacidad de adquirir varias formas.

Es un concepto muy importante en cuanto a orientación a objetos y a lo que a la herencia se refiere. Consiste en que un objeto dependiendo de la llamada a un método u otro tiene un comportamiento diferente.

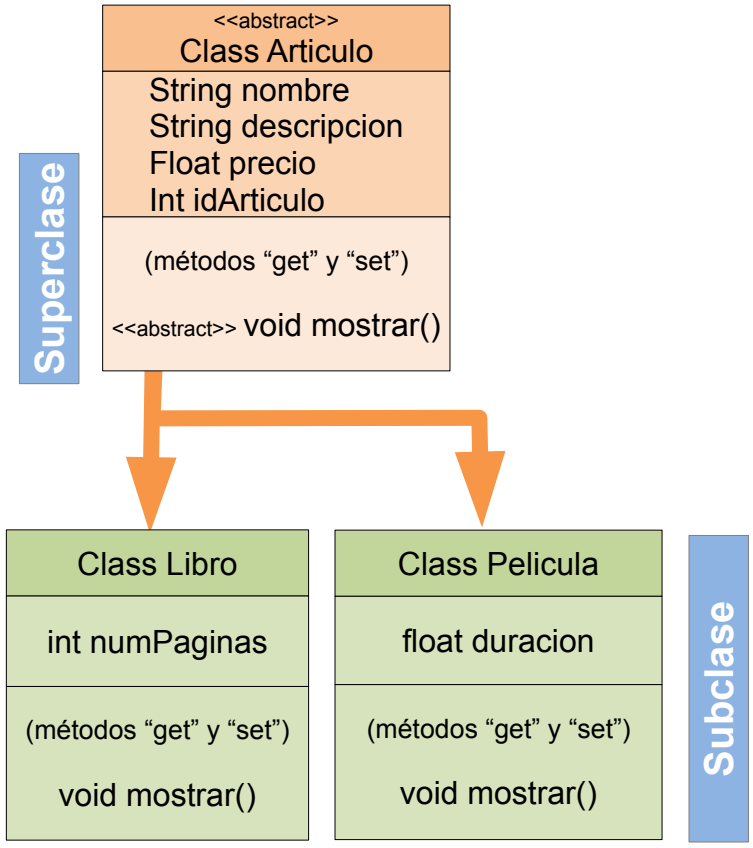
Este comportamiento puede ser determinado en tiempo de compilación (cuando aún no hemos ejecutado el programa), es lo que en el *[apartado 4.5 Sobrecarga]* vimos, de tal manera que dependiendo de la cantidad, tipo y orden de los parámetros el compilador puede determinar a que método estamos ligando la llamada. Otra manera que existe es mediante la ligadura dinámica, la cual a diferencia del caso anterior se determina en tiempo de ejecución

6.5.1 Ligadura dinámica.

Esto sucede cuando una subclase sobrescribe métodos de la superclase, de tal manera que la misma llamada a un método conlleve a la ejecución de métodos diferentes dependiendo de la instancia desde la que se llame. Pero como se podrá observar esto no se sabe hasta que se está en tiempo de ejecución, por tanto es ahí cuando se determina el método a llamar.

Para poder ilustrar de una manera más práctica los conceptos de polimorfismo y ligadura dinámica vamos a ver el siguiente ejemplo.

Vamos a suponer el siguiente esquema:



Como podemos observar tenemos unos métodos comunes, y entre ellos sobrescribimos el método "mostrar" de la superclase en cada una de las subclases debido a que es abstracto.

La idea principal es que cuando yo creo una instancia de la clase "Libro" o la clase "Pelicula" y llamo a su método "mostrar" respectivamente cada una de ellas responde con el contenido de su método, pero, ¿que sucede si llamo a ese método desde la clase "Articulo"?, en eso consiste el Polimorfismo, hacer que un mismo "Articulo" adquiera diferentes formas en base a la instancia que contiene. Para poder probar esto crearemos las clases arriba indicadas y además crearemos un ejemplo con una clase "Articulo" en la cual probamos una instancia de la clase "Libro" y otra de la clase "Pelicula".

El código correspondiente a "Articulo.java":

```
public abstract class Articulo {
	private int idArticulo;
	private String nombre,descripcion;
	private float precio;
	public Articulo(int id,String nom,String des,float p){
		idArticulo=id;
		nombre=nom;
		descripcion=des;
		precio=p;
	}
	public int getIdArticulo() {
```

```
		return idArticulo;
	}
	public void setIdArticulo(int idArticulo) {
		this.idArticulo = idArticulo;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getDescripcion() {
		return descripcion;
	}
	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}
	public float getPrecio() {
		return precio;
	}
	public void setPrecio(float precio) {
		this.precio = precio;
	}
	public abstract void mostrar();
}
```

El código correspondiente a “Libro.java”:

```
public class Libro extends Articulo {
	private int numPaginas;

	public Libro(int id, String nom, String des, float p,int pags) {
		super(id, nom, des, p);
		numPaginas=pags;
	}
	public int getNumPaginas() {
		return numPaginas;
	}
	public void setNumPaginas(int numPaginas) {
		this.numPaginas = numPaginas;
	}

	@Override
	public void mostrar() {
		System.out.println("id:"+getIdArticulo()+"\tNombre:"+getNombre()
+"("+numPaginas+"pags.)\tPrecio:"+getPrecio()+" euros");
	}
}
```

El código correspondiente a “Pelicula.java”:

```
public class Pelicula extends Articulo{
```

```
	private float duracion;

	public Pelicula(int id, String nom, String des, float p,float dur) {
		super(id, nom, des, p);
		duracion=dur;
	}

	public void setDuracion(float dur){
		duracion=dur;
	}
	public float getDuracion(){
		return duracion;
	}

	@Override
	public void mostrar() {
		System.out.println("id:"+getIdArticulo()+"\tNombre:"+getNombre()
+"("+duracion+"min.)\tPrecio:"+getPrecio()+" euros");
	}
}
```

Una vez tenemos esto podemos proceder a realizar el ejemplo en un “main”:

```
	public static void main(String[] args){
		Articulo art=new Pelicula(1,"Nueva peli","Descripcion pelicula",
					45.7f,1.5f);
		art.mostrar();
		art=new Libro(12,"Nuevo Libro","Descripcion libro",
				45.7f,120);
		art.mostrar();
	}
```

Cuando ejecutamos el código anterior podemos ver que aunque “a” es una clase “Articulo” dependiendo de la instancia que contiene su comportamiento es diferente, mostrándonos los datos de una película o un libro respectivamente, en eso consiste el polimorfismo y la ligadura dinámica.

Ahora veamos que sucede en el siguiente “main”:

```
	public static void main(String[] args){
		Articulo art=new Pelicula(1,"Nueva peli","Descripcion pelicula",
					45.7f,1.5f);
		art.getDuracion();
		art.mostrar();
	}
```

El código anterior nos produciría un error, ya que aunque el “mostrar” si funcionaría, el método “getDuracion” aunque pertenezca a “Pelicula” no va a ser encontrado porque no existe en la clase “Articulo”, por tanto no se puede realizar ningún tipo de ligadura.

Por tanto resumiendo para que se den las condiciones de polimorfismo y ligadura dinámica se han de cumplir las siguientes condiciones:

- Que haya herencia.

- Que haya sobrescritura de métodos de la supercalse.
- Definir un objeto con la superclase e instanciarlo con la subclase.

Ejercicio Auto-evaluación A6.5: *Agrega al ejemplo anteriormente visto de Artículos un nuevo artículo "Ordenador", este debe tener una RAM de tipo entero y una CPU de tipo float.*

Ejercicio Auto-evaluación A6.6: *Con el ejemplo y el ejercicio anterior realizar una clase "Cesta" que va a realizar las funciones de cesta de la compra donde meteremos artículos, por tanto tendremos un array estático o dinámico de "Articulo", ayúdate de lo visto en el tema anterior para realizarlo. La cesta va a tener un método "agregarArticulo" al cual se le pasa un articulo y lo agrega a la cesta, y también tendrá un método "total" que nos mostrará un desglose de todos los artículos contenidos en la cesta donde nos muestra el "id", el "nombre" y el "precio" y al final del desglose la suma total del precio de los artículos. Realizar un ejemplo en el que se introducen varios artículos en una cesa y se calcula el total.*

6.5.2 Comprobación de tipos.

Existe una manera de poder evitar el fallo anterior en el que llamábamos al método "getDuracion" de "Pelicula", y esta es mediante la comprobación de tipos. Es posible comprobar el tipo del la instancia contenida en "Articulo" en tiempo de ejecución de tal manera que si es una "Pelicula" podemos introducirla dentro de una clase "Pelicula" y tener acceso al método. Para averiguar el tipo usamos la palabra reservada "***instanceof***" que viene a ser algo así como "*instancia de*". La sintaxis es la siguiente:

*Clase **instanceof** tipoInstancia*

Eso nos devolverá "true" si es del tipo de instancia que comprobamos o "false" en el caso contrario. Por último sería necesario hacer un casting al tipo de objeto concreto para poder llamar a sus émtodos. Veamos un ejemplo en el que comprobamos si es un "Libro" o es una "Pelicula" y llamamos al método concreto:

```
public static void main(String[] args){
      Articulo art=new Pelicula(1,"Nueva peli","Descripcion pelicula",
                                    45.7f,1.5f);
      if(art instanceof Pelicula){
            Pelicula p=(Pelicula)art;
            System.out.println("Es una pelicula y dura:"+p.getDuracion());
      }
      if(art instanceof Libro){
            System.out.println("Es un Libro y su número de páginas es:"+
                        ((Libro)art).getNumPaginas());
      }
}
```

Como se observa comprobamos el tipo de instancia y si es de una o de otra hacemos un casting y llamamos a sus métodos concretos. En “Pelicula” realizamos un casting a través de un nuevo objeto y en “Libro” lo hacemos directamente, realmente estamos haciendo lo mismo pero se ha decidido así para que el lector vea dos maneras diferentes de realizar la misma acción.

Ejercicio Auto-evaluación A6.7: *Utilizando lo visto en este apartado y la “Cesta” realizada en el ejercicio A6.6, realiza sobre los artículos de tipo “Ordenador” un 30% de descuento sobre el precio. Ponlo a prueba con un ejemplo.*

6.6 Ejercicios Propuestos

1. Crear una clase Huevo que tiene dos clases internas Clara y Yema respectivamente. La Clara tiene un atributo proteínas de tipo entero, y la Yema tiene un atributo color. El huevo va a tener dos métodos, uno que es dameClara que devuelve la Clara, y otro que es dameYema.

2. Construir una clase Math2 que va a tener los mismos métodos que la clase Math de Java. La clase Math2 va a tener un método llamado sumaArray que recibe como parámetro un array de enteros unidimensional y devuelve un entero con la suma de todos los números del array convertidos a positivo. Agregar también un método min que devuelve el menor de los números del array, otro max que devuelve el mayor de los números del array.

3. Crear una clase DNISinLetra que tiene como atributo un String. Cuando creemos un DNI debe pasarse como parámetro. Debe tener un método que nos sirva para comprobar que el DNI es correcto, que tiene 8 números, en el caso de que el DNI sea incorrecto nos pondrá el DNI como “00000000”. Después crearemos una clase DNIConLetra que tiene los mismos atributos que DNISinLetra, pero cuando introducimos un DNI válido al final del mismo nos incluye la letra que corresponda. (Para ver la correspondencia de letras mirar el ejercicio propuesto 6 del Tema 3).

4. Crear una clase Factura que recibe la cesta del ejercicio de Auto-evaluación 6.6 y tiene como atributos un idFactura de tipo entero, empresa que es el nombre de la empresa de tipo cadena, e idCliente de tipo entero. La Factura tendrá un método llamado imprimirFactura que nos servira para mostrar datos de la factura por pantalla. A continuación crear una clase llamada FacturaDetalle que nos muestra lo mismo que Factura pero con los artículos detallados. Crear también una clase FacturaIVA, que tendrá un atributo IVA de tipo float, y que nos muestra lo mismo que Factura pero nos muestra el total sin iva y a continuación con el iva incluido.

5. Crear una clase Animal que tiene un atributo nombre que es el nombre del animal. La clase Animal va a tener dos métodos, uno que se llama presentarse() que imprime por pantalla “Hola, me llamo “ y el nombre del animal, por ejemplo “Hola, soy un TOBY”, y el otro método será hablar() que nos mostrará lo que dice el animal, por ejemplo un perro diría “GUAU”. Crear 4 clases de animales, una clase Perro que dice “GUAU”, una clase Gato que dice “MIAU”, una clase Pajaro que dice “PIO” y una clase Leon que dice “GRRR”. Por último crear una clase Zoologico que tendrá un nombre y que tiene un conjunto de animales de varios tipos y con diferentes nombres, la clase Zoologico tendrá un método llamado visitar() que hace que cada uno de los animales se presente y a continuación hable.

6. Usando el ejercicio anterior agregar al Zoologico un método llamado calificacion que muestra por pantalla “PELIGROSO” si el Zoologico tiene más de dos leones e “INOFENSIVO” en el caso contrario.

7. Realizar un sistema de envío de mensajes. Vamos a tener una clase Mensaje *que tiene un atributo mensaje de tipo String y un atributo destinoremitente de tipo String que es destinatario al que lo enviamos o el remitente que nos ha enviado y los métodos “String enviarMensaje()” que muestra por pantalla el mensaje y lo devuelve como parámetro, y un “recibirMensaje(String)” que muestra el mensaje recibido por pantalla. Después tendremos 3 clases diferentes, una clase SMS, otra MMS, y otra Email. La clase SMS solo puede enviar texto. La clase MMS puede enviar texto y a parte imágenes (que será otro atributo de texto a parte). La clase Email puede enviar texto y archivos (que será un atributo de texto y un atributo entero).*

6.7 Solución a los ejercicios de Auto-evaluación

Solución Auto-evaluación A6.1:

El "Triangulo.java":

```
public class Triangulo extends Figura {
	protected float base, altura;

	public Triangulo(String nombre, String color,float b, float a)
	{
		super(nombre, color);
		this.base=b;
		this.altura=a;
	}

	public void mostrar()
	{
		super.mostrar();
		System.out.println("Soy el triángulo:"+nombre+" de color "+color
				+" y mi área es:"+calcular_area());
	}

	public float calcular_area()
	{
		return (base*altura)/2;
	}
}
```

El "Cuadrado.java":

```
public class Cuadrado extends Figura{
	protected float lado;

	public Cuadrado(String nombre, String color,float l)
	{
		super(nombre, color);
		this.lado=l;
	}

	public void mostrar()
	{
		super.mostrar();
		System.out.println("Soy el cuadrado:"+nombre+" de color "+color
				+" y mi área es:"+calcular_area());
	}

	public float calcular_area()
	{
		return (lado*lado);
	}
}
```

Solución Auto-evaluación A6.2:

La clase “Vehiculo.java”:

```
public class Vehiculo {
	private String color;
	private String marca;

	public Vehiculo(String m, String c){
		marca=m;
		color=c;
	}

	public void setColor(String c){
		color=c;
	}

	public String getColor(){
		return color;
	}

	public void setMarca(String m){
		marca=m;
	}

	public String getMarca(){
		return marca;
	}

	public void mostrar(){
		System.out.println("Marca:"+marca+"\tColor:"+color);
	}
}
```

La clase “Coche.java”:

```
public abstract class Coche extends Vehiculo {
	private int numRuedas;

	public Coche(String m,String c,int r){
		super(m,c);
		numRuedas=r;
	}

	public void setNumRuedas(int r){
		numRuedas=r;
	}

	public int getNumRuedas(){
		return numRuedas;
	}

	public void mostrar(){
		System.out.println("COCHE\n***************");
		super.mostrar();
		System.out.println("Num.Ruedas:"+numRuedas);
```

```
	}
}
```

La clase “Barco.java”:

```
public class Barco extends Vehiculo {
	private float eslora;

	public Barco(String m,String c,float e){
		super(m,c);
		eslora=e;
	}

	public void setEslora(float e){
		eslora=e;
	}

	public float getEslora(){
		return eslora;
	}

	public void mostrar(){
		System.out.println("BARCO\n***************");
		super.mostrar();
		System.out.println("Eslora:"+eslora);
	}
}
```

La clase “Avion.java”:

```
public class Avion extends Vehiculo {
	private int helices;

	public Avion(String m,String c,int h){
		super(m,c);
		helices=h;
	}

	public void setHelices(int h){
		helices=h;
	}

	public int getHelices(){
		return helices;
	}

	public void mostrar(){
		System.out.println("AVIÓN\n***************");
		super.mostrar();
		System.out.println("Num.Helices:"+helices);
	}
}
```

Solución Auto-evaluación A6.3:

Como el método “calcularSueldo” va a depender de cada empleado y en cada uno de

ellos debe implementarse, es lógico pensar que este debía ser abstracto. El atributo "horas" podríamos haberlo situado en "Empleado", pero siendo previsores, y pensando que quizá el sueldo de otros futuros puestos de empleados no sea calculado a través de las horas (que pueda ser por días, semanas o incluso fijo cada mes), por tanto se ha optado por situar este atributo en las subclases. Dicho esto veamos el resultado.

La clase "Empleado.java":

```
// Mi clase Persona la tengo en el paquete "Comun",
// en el caso del lector puede ser en otro diferente.
import Comun.Persona;

public abstract class Empleado extends Persona{
	protected int id;
	protected float sueldo;
	public Empleado(String n, int e,int id) {
		super(n, e);
		this.id=id;
		sueldo=0;
	}

	public abstract void calcularSueldo();
}
```

La clase "Secretario.java":

```
public class Secretario extends Empleado{
	private int horas;

	public Secretario(String n, int e, int id,int horas) {
		super(n, e, id);
		this.horas=horas;
	}

	@Override
	public void calcularSueldo() {
		System.out.println("El sueldo del empleado "+id+" es:"+horas*7+"
euros.");
	}
}
```

La clase "Comercial.java":

```
public class Comercial extends Empleado{
	private int horas;
	private float km;

	public Comercial(String n, int e, int id,int horas,int km) {
		super(n, e, id);
		this.horas=horas;
		this.km=km;
	}

	@Override
	public void calcularSueldo() {
		System.out.println("El sueldo del empleado "+id+" es:"+
```

```
(horas*6+km*0.30)+" euros.");
	}
}
```

El ejemplo que se pide en el ejercicio:

```
	public static void main(String[] args) {
		Secretario s=new Secretario("Paco",30,4,160);
		Comercial c=new Comercial("Juan",25,7,150,2000);

		s.calcularSueldo();
		c.calcularSueldo();
	}
```

Solución Auto-evaluación A6.4:

```
public class Rectangulo extends Figura implements Dibujable,Calculable{
	protected float base, altura;

	public Rectangulo(String nombre, String color,int posicion,float b, float
a)
	{
		super(nombre, color,posicion);
		this.base=b;
		this.altura=a;
	}
	@Override
	public void mostrar()
	{
		System.out.println("Soy el rectángulo:"+nombre+" de color "+color
				+" y mi área es:"+calcularArea());
	}
	@Override
	public float calcularArea()
	{
		return (base*altura);
	}
	@Override
	public void setPosicion(int pos) {
		posicion=pos;
	}
	@Override
	public void dibujar() {
		String desplaza="";
		for(int i=0;i<posicion;i++){
			desplaza=desplaza+"  ";
		}
		for(int i=0;i<altura;i++){
			System.out.print(desplaza);
			for(int j=0;j<base;j++){
				System.out.print("* ");
			}
			System.out.println();
		}
	}
```

```
	public static void main(String[] args){
		Rectangulo r1=new Rectangulo("r1","Azul",1,10,5);
		r1.mostrar();
		r1.dibujar();
		r1.setPosicion(15);
		r1.dibujar();
	}
}
```

Solución Auto-evaluación A6.5:

```
public class Ordenador extends Articulo{
	private int RAM;
	private float CPU;

	public Ordenador(int id, String nom, String des, float p,int ram,float
cpu) {
		super(id, nom, des, p);
		this.setRAM(ram);
		this.CPU=cpu;
	}

	@Override
	public void mostrar() {
		System.out.println("id:"+getIdArticulo()+"\tNombre:"+getNombre()
+"("+CPU+"Mhz.)\tPrecio:"+getPrecio()+" euros");
	}

	public int getRAM() {
		return RAM;
	}

	public void setRAM(int rAM) {
		RAM = rAM;
	}
	public float getCPU() {
		return CPU;
	}

	public void setCPU(float cPU) {
		CPU = cPU;
	}
}
```

Solución Auto-evaluación A6.6:

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

public class Cesta {
	private ArrayList<Articulo> miCesta;

	public Cesta(){
		miCesta=new ArrayList<Articulo>();
```

```
	}
	public ArrayList<Articulo> getMiCesta() {
		return miCesta;
	}

	public void setMiCesta(ArrayList<Articulo> miCesta) {
		this.miCesta = miCesta;
	}

	public void agregarArticulo(Articulo a){
		miCesta.add(a);
	}

	public float total(){
		float totalCompra=0;
		Iterator<Articulo> i=miCesta.iterator();
		Articulo a;
		while(i.hasNext()){
			a=i.next();
			a.mostrar();
			totalCompra=totalCompra+a.getPrecio();
		}
		System.out.println("TOTAL-> "+totalCompra+" euros.");
		return totalCompra;
	}

	public static void main(String[] args){
		Cesta c=new Cesta();
		c.agregarArticulo(new Pelicula(1,"Nueva peli","Descripcion
pelicula",45.7f,1.5f));
		c.agregarArticulo(new Libro(12,"Nuevo Libro","Descripcion
libro",45.7f,120));
		c.agregarArticulo(new Ordenador(145,"Nuevo ordenador","Descripcion
ordenador",100f,3,1.5f));
		c.total();
	}
}
```

Solución Auto-evaluación A6.7:

Los cambios a realizar en “Cesta” en este caso están en el método “total”, simplemente debemos comprobar cada tipo y aplicarle el descuento.

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

public class Cesta {
	private ArrayList<Articulo> miCesta;
	public Cesta(){
		miCesta=new ArrayList<Articulo>();
	}

	public void agregarArticulo(Articulo a){
		miCesta.add(a);
```

```
	}

	public void total(){
		float totalCompra=0;
		Iterator<Articulo> i=miCesta.iterator();
		Articulo a;
		while(i.hasNext()){
			a=i.next();
			if(a instanceof Ordenador){
				a.setPrecio(a.getPrecio()-a.getPrecio()*0.1f);
			}
			if(a instanceof Libro){
				a.setPrecio(a.getPrecio()-a.getPrecio()*0.25f);
			}
			a.mostrar();
			totalCompra=totalCompra+a.getPrecio();
		}
		System.out.println("TOTAL-> "+totalCompra+" euros.");
	}

	public static void main(String[] args){
		Cesta c=new Cesta();
		c.agregarArticulo(new Pelicula(1,"Nueva peli","Descripcion
pelicula",45.7f,1.5f));
		c.agregarArticulo(new Libro(12,"Nuevo Libro","Descripcion
libro",45.7f,120));
		c.agregarArticulo(new Ordenador(145,"Nuevo ordenador","Descripcion
ordenador",100f,3,1.5f));
		c.total();
	}
}
```