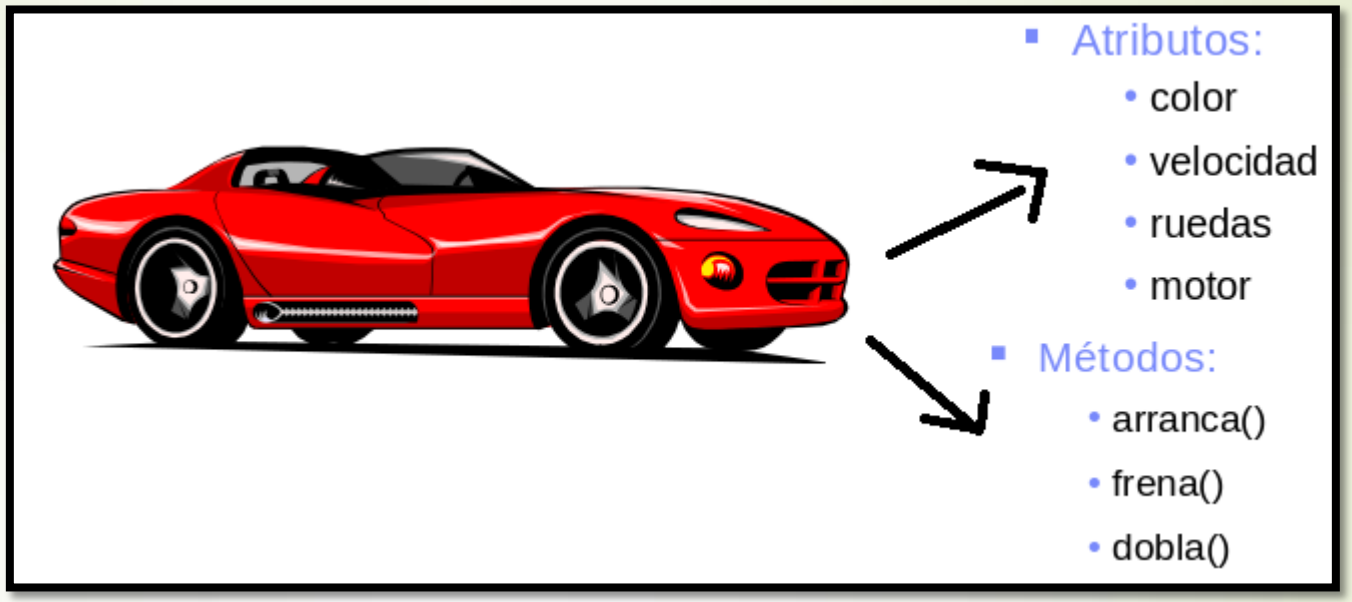
Desarrollo de Aplicaciones Web y Multiplataforma: **Programación**

DOCENTE: Daniel López Lozano

escuelaartegranada

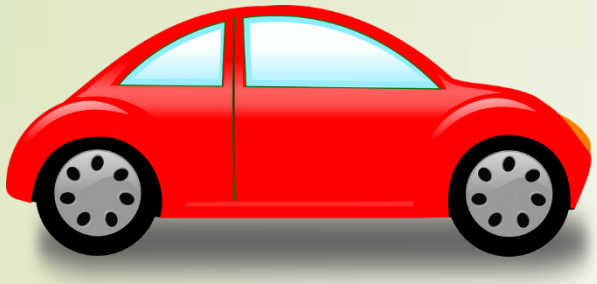


Tema 5.
Introducción al Diseño Orientado a Objetos en Java

Índice de contenidos

- A pesar de utilizar clases y objetos en nuestros programas el software que hemos construido **no están orientado a objetos**.
- El objetivo es descomponer el problema en problemas más pequeños, que sean fáciles de manejar y mantener.
- Fijándonos en cuál es el escenario del problema e intentar reflejarlo en nuestro programa.
- Se trata de trasladar la visión del mundo real a nuestros programas.
- La **Programación Orientada a Objetos** es más cercana a nuestra forma de analizar, entender y resolver los problemas.

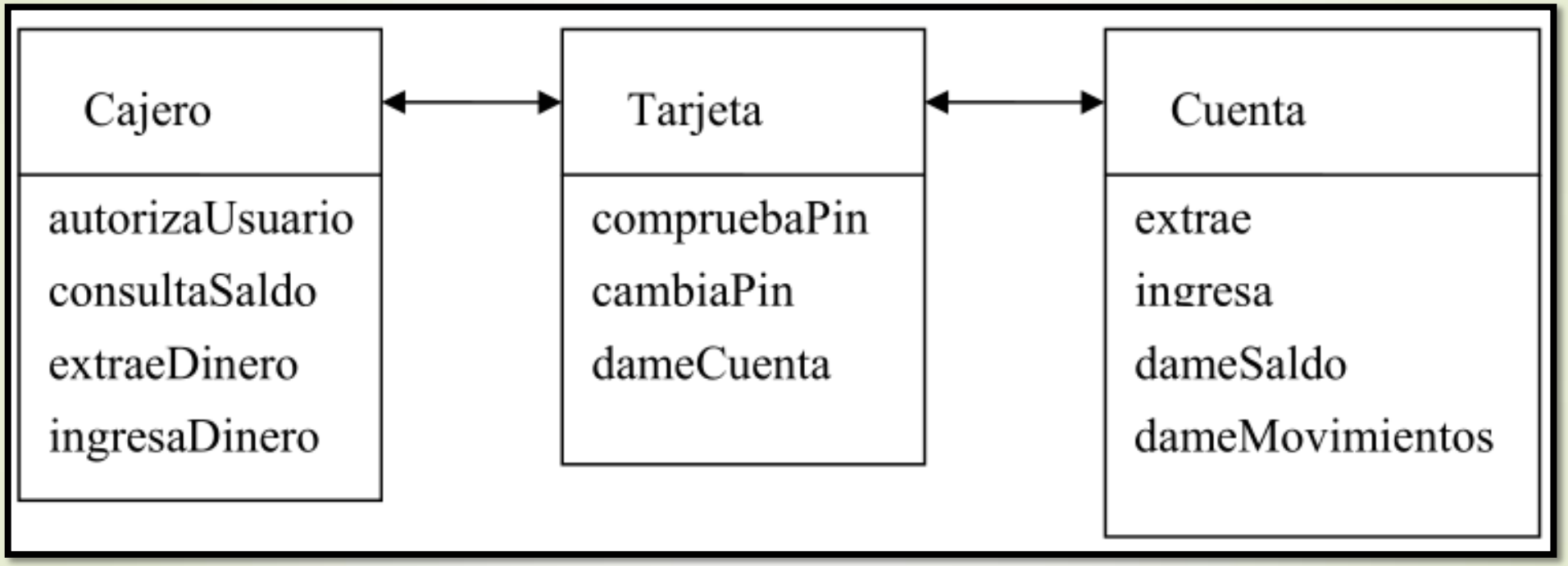
- Las técnicas y definiciones que se usan en un programa orientado a objetos pretenden conseguir las siguientes características:
- **Abstracción.** Consiste en ver a algo como un todo sin saber cómo está formado internamente.
- Las personas gestionan la complejidad a través de la abstracción, por ejemplo para alguien es difícil entender todos los componentes y circuitos de un televisor y como trabajan.
- Sin embargo es más fácil conocerlo como un todo, como un televisor, sin pensar en sus detalles o partes internas.



- Vamos a practicar la abstracción, modelando distintas situaciones como se haría en POO.
- Nos vamos a fijar en sus partes esenciales y que pueden hacer (**acciones** o **capacidades**) más que realmente como se hacen o su funcionamiento interno.
- Las partes esenciales son **características**.
- Las acciones son capacidades
  - Cuenta corriente
  - Televisión.
  - Reproductor de música.
  - Procesador de textos.
  - Personaje videojuego.
  - Enrutador de Internet.
  - Agenda de contactos.

- La abstracción no es solo un proceso para convertir elementos de la realidad para poder programarlos.
- Es aplicable a los propios componentes de un sistema informático y por tanto es la esencia de cualquier desarrollo.
- Si abstraemos estamos pensando orientado a objetos directamente. Por ejemplo:
- ¿Cuáles serían las características de una **Ventana (de un sistema operativo tipo Windows o similar)**? ¿cuáles serían las capacidades? Piensa en las propiedades y en el comportamiento de una ventana de cualquier programa.

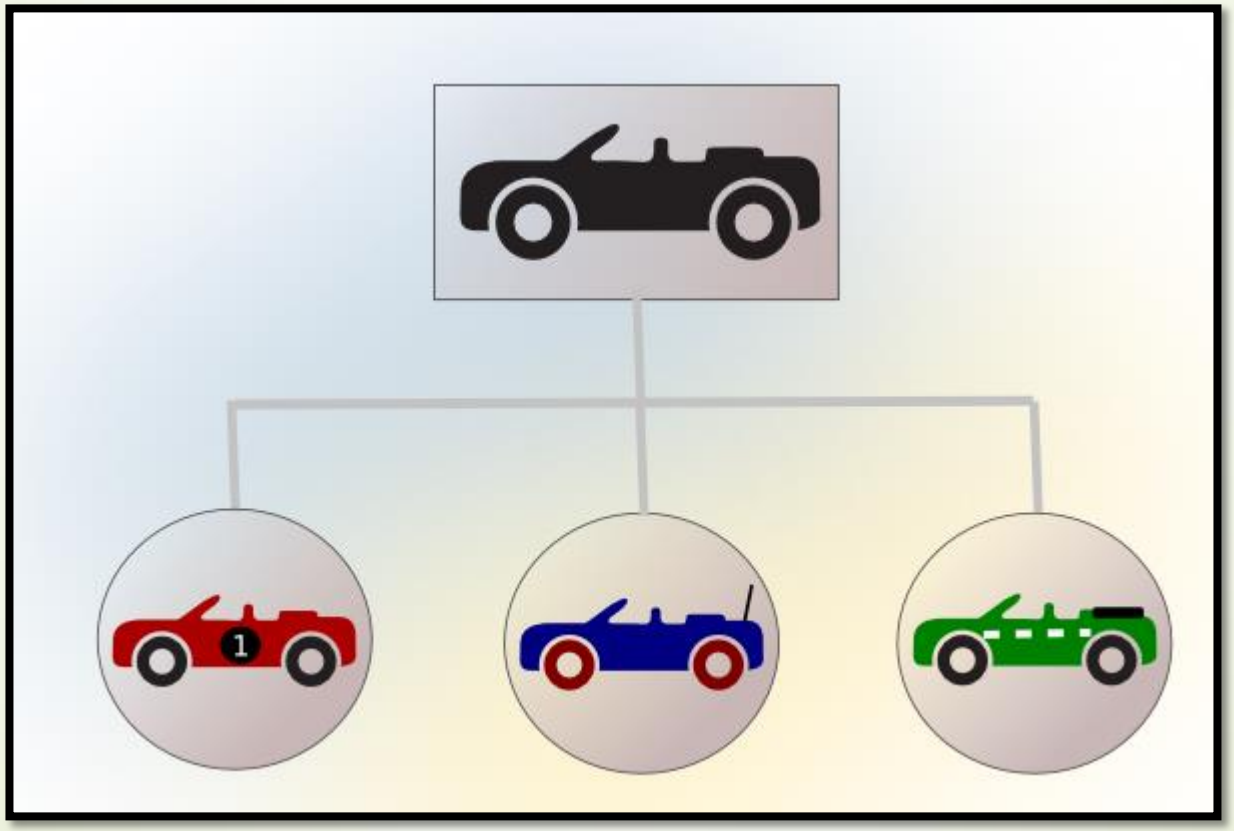
Diseño orientado a objetos



- El mecanismo para conseguir la abstracción en la POO son **las clases**.
- **Encapsulamiento.** Hablamos de ocultar información, significa que sólo se debe mostrar los detalles esenciales de un objeto, mientras que los detalles no esenciales se deben ocultar.
- Para conseguir la encapsulación existen modificadores de visibilidad de variables y métodos cuyo objetivo es definir claramente como usar una clase (interfaz).
- **Herencia.** El concepto fundamental de la herencia es el proceso en el que un objeto adquiere características de otro objeto.

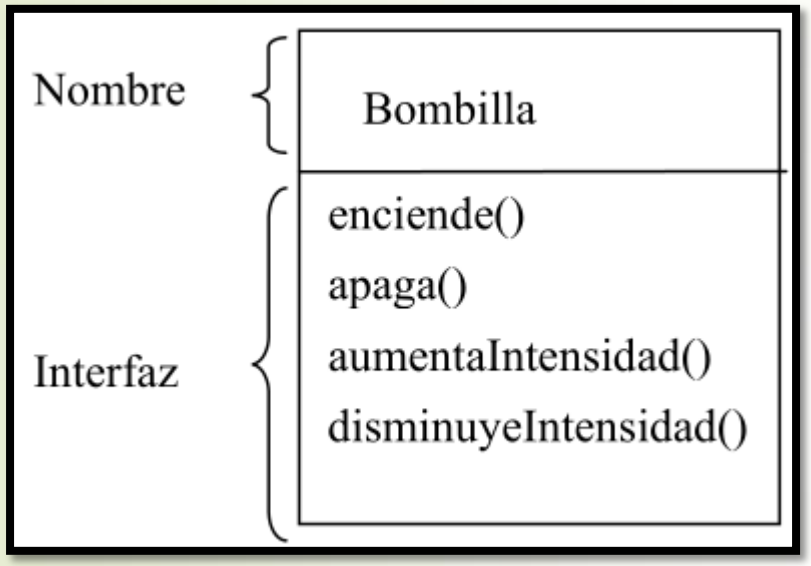
- La herencia es un mecanismo muy potente y es el principal atractivo de la orientación objetos.
- Establece un mecanismo sencillo y rápido para reutilizar y extender software.
- **Polimorfismo.** Es la capacidad de un objeto para comportarse de diferentes formas. No se puede aplicar este concepto sin la herencia.
- La herencia se puede usar para definir que varios objetos pertenecen a un mismo grupo, pero cada uno con sus características propias.

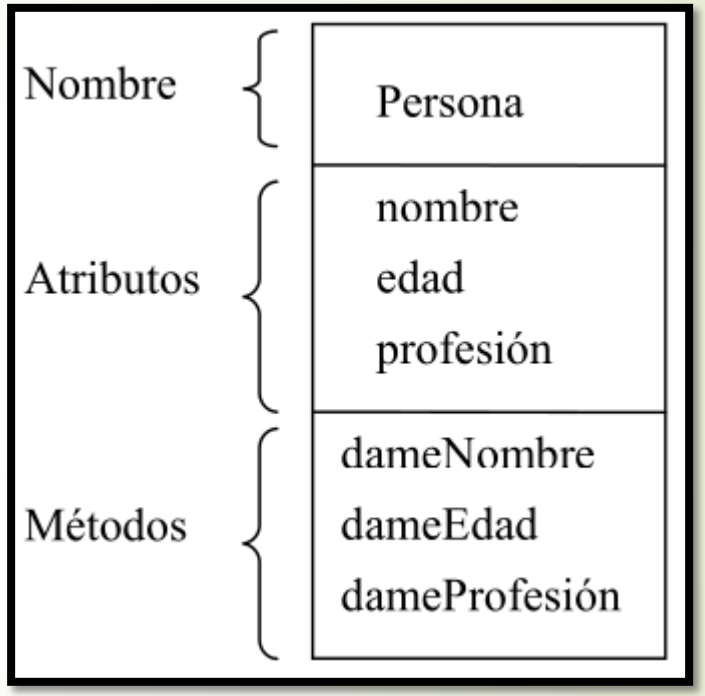
- El polimorfismo expresa dicho comportamiento aportando dos funcionalidades muy importantes que son la **compatibilidad y genericidad**.



- Los objetos son elementos que definen dos características: **estado y comportamiento**.
- El **estado** de un objeto viene definido por una serie de características. En el caso de tener un objeto perro, su estado estaría definido por **su raza, color de pelo, tamaño**, etc.
- El comportamiento viene definido por las acciones que pueden realizar los objetos, por ejemplo, en el caso del perro su comportamiento sería: **saltar, correr, ladrar**, etc.
- Los valores que tengan las características de un objeto son los que lo diferencian de otro distinto.

Comportamiento de un objeto





- Una clase es una **plantilla, molde o modelo** para crear objetos.
- La clase la vamos a utilizar para definir la estructura de un objeto, es decir, estado (**atributos**) y comportamiento (**métodos**).
- Para diferenciar y relacionar clases y objetos podemos pensar que un molde para hacer galletas sería una clase, y las galletas que hacemos a partir de ese molde ya son objetos concretos creados a partir de las características definidas por el molde.
- Los objetos son instancias de una clase y los valores de los atributos diferencian unas instancias de otras.

Los objetos son instancias de clases

Clase	Objetos		
Persona	Persona	Persona	Persona
nombre edad profesión	José 23 carpintero	Laura 37 administrativa	Cristina 19 estudiante

A ver si tenemos claro la diferencia entre clase y objeto

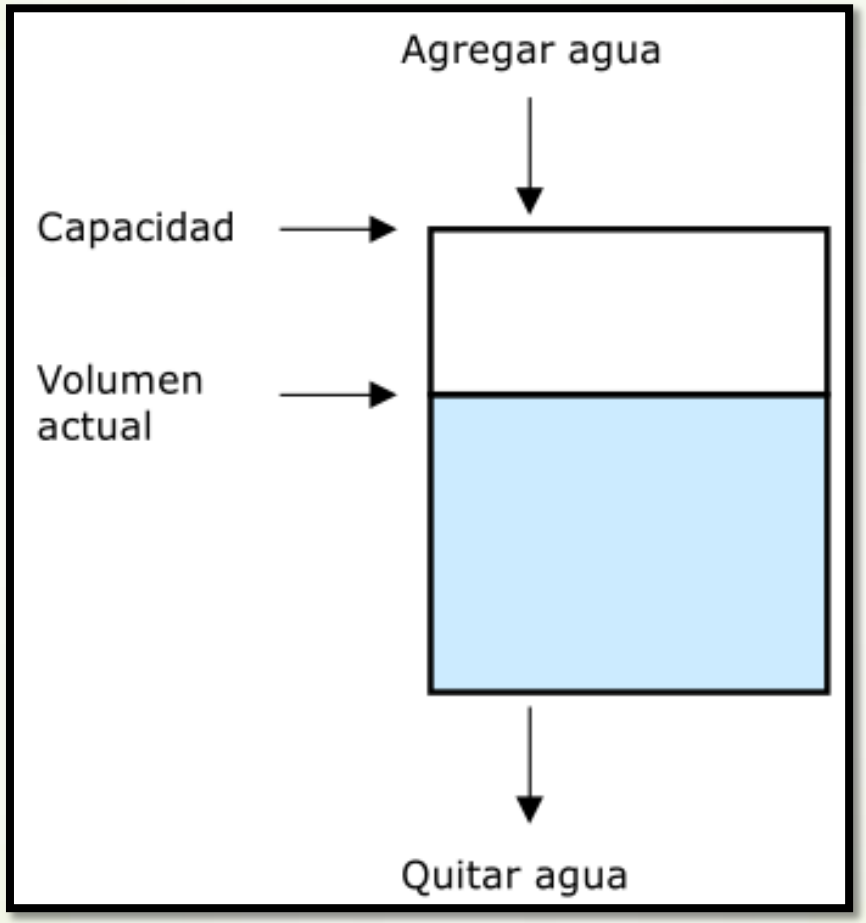
- Paula
- Gardield
- Perro
- Mineral
- Caballo
- Tomas
- Silvestre
- Snoopy
- Pato Lucas

- Snoopy
- Gato
- Animal
- Alumno
- Pegaso
- Ayudante de Santa Claus
- Cuarzo
- Laika
- Ejecutivo

```
class MiClase{

    //Constructor de la Clase

    //atributos de la clase

    //métodos de la clase

}
```

- Los atributos son variables que describen como es el objeto y son accesibles desde cualquier punto de la clase, o también llamado el contexto del objeto.
- Los métodos en este caso hacen uso de los atributos del objeto para hacer su cometido y definen que se puede hacer con el objeto.

Ejemplo modelodo de clases



El modelo de clases está compuesto por:

- Elementos tangibles (objetos).
  - Depósitos de agua.
- Atributos (campos).
  - Capacidad (m^3)
  - Volumen actual (m^3)
- Capacidades o Responsabilidades (operaciones).
  - Agregar agua (m^3)
  - Quitar agua (m^3)
- Limitaciones de diseño (contrato del objeto).
  - La capacidad del depósito debe ser mayor que cero y no puede modificarse.
  - El volumen actual no puede ser negativo.
  - No se puede quitar agua de un depósito vacío ni desbordarlo.

Programación de la clase Deposito

```java
public class Deposito {
    double capacidad;
    double volumen_actual;
    String id_deposito;

    //Constructor
    public Deposito(String id,double c){
        capacidad=c;
        id_deposito=id;
        volumen_actual=0;
    }

    //Selectores o getters
    public double obtenerVolumenActual(){
        return volumen_actual;
    }

    public double obtenerCapacidad(){
        return capacidad;
    }

    public String obtenerId(){
        return id_deposito;
    }
```

```java
public boolean estaVacio(){
    return volumen_actual==0;
}

//Modificadores o setters

public void agregarVolumen(double volumen){
    if(volumen_actual+volumen<=capacidad_max){
        volumen_actual+=volumen;
    }else{
        System.out.println("No se puede añadir tanto líquido");
    }
}

public void retirarVolumen(double volumen){
    if(volumen_actual-volumen>=0){
        volumen_actual-=volumen;
    }else{
        System.out.println("No hay suficiente líquido");
    }
}
```

Implementación de la clase Deposito

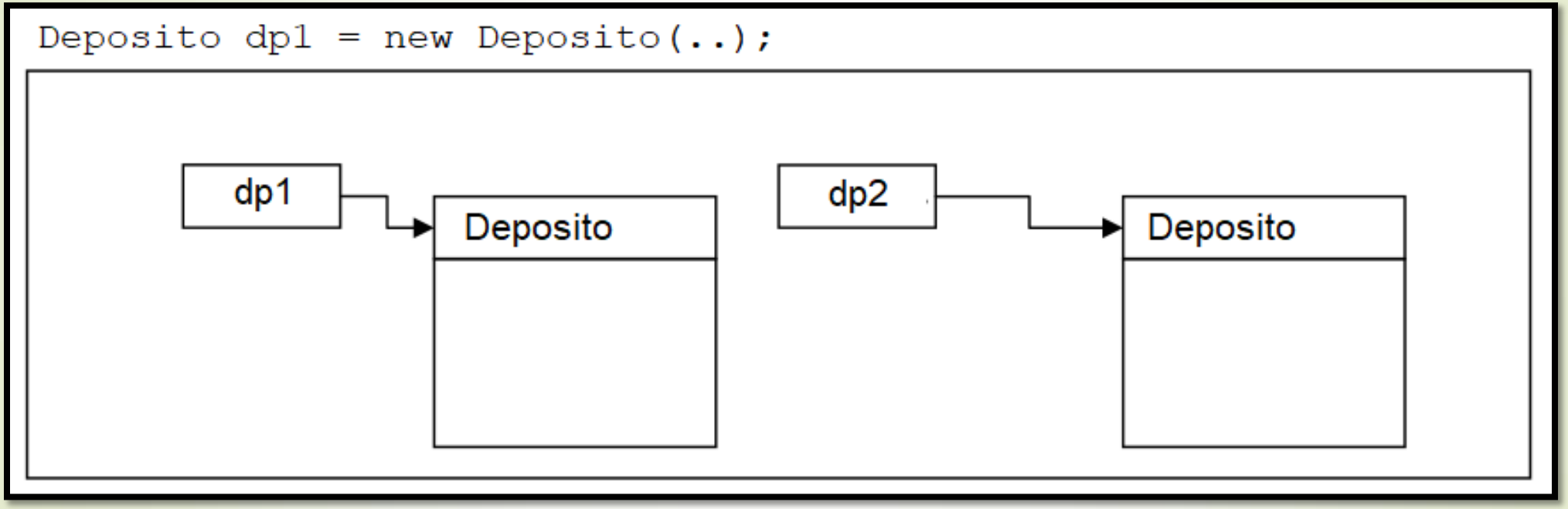
```
//Visualizadores

public void info_deposito(){
    System.out.println("============================");
    System.out.println("|ID Deposito: "+id_deposito+"m3    |");
    System.out.println("|Capacidad total: "+capacidad+"m3 |");
    System.out.println("|Volumen actual: "+volumen_actual+"m3   |");
    System.out.println("============================");

}
```

Prueba clase Deposito

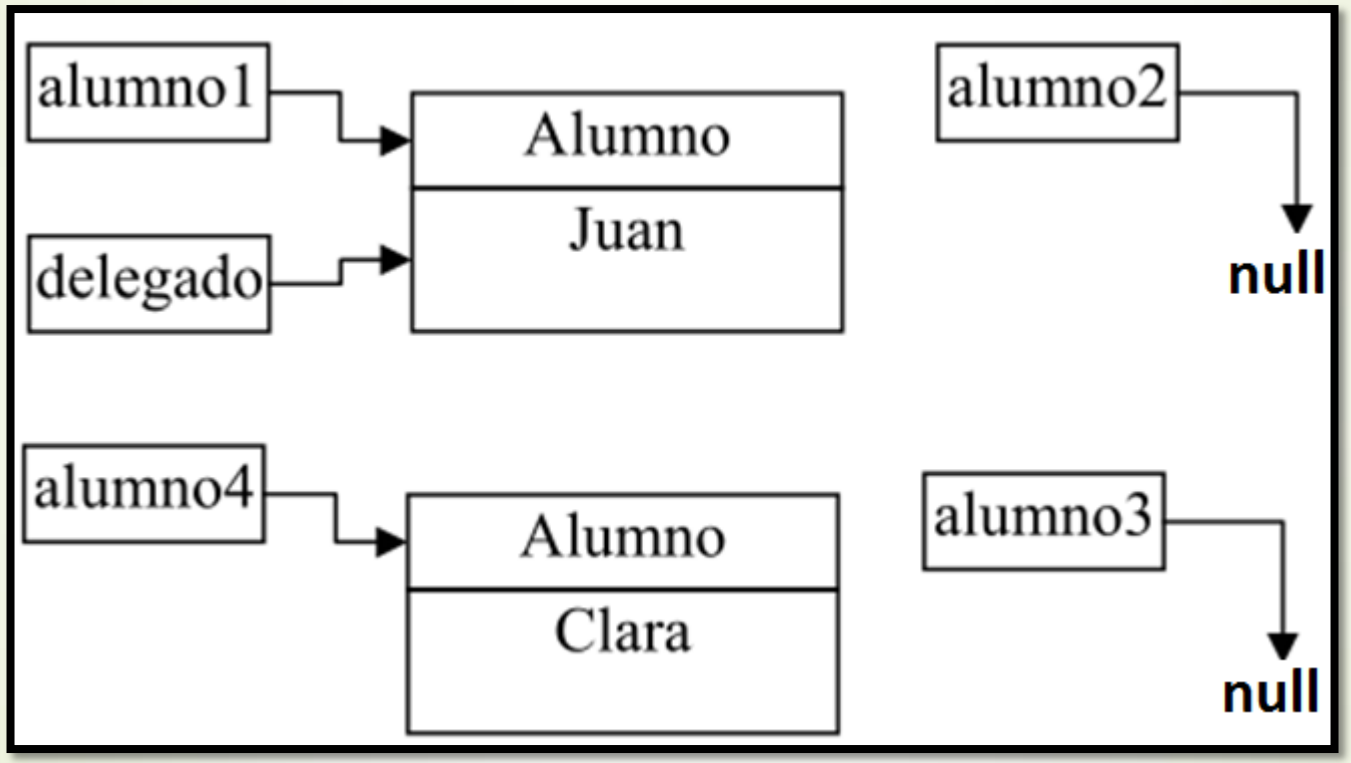
```
public class PruebaDeposito{
    public static void main(String[] args) {
        Deposito dp=new Deposito("ID_GRA1", 300);
        dp.info_deposito();
        dp.agregarAgua(100);
        dp.info_deposito();
        dp.sacarAgua(40);

        if(dp.estaVacio()){
            System.out.println("El deposito esta vacio");
        }else{
            System.out.println("El deposito no esta vacio");
        }
        //Otro deposito con las mismas caracteristicas
        //Pero distinto del anterior
        Deposito dp2=new Deposito("ID_GRA2", 500);
        dp2.agregarAgua(100);
        if(dp2.obtenerVolumenActual()>=100){
            System.out.println("El deposito "+
                                dp2.obtenerId()+
                                " tiene al menos 100");
        }
```

- Dentro de la funcionalidad que define una clase a partir de sus métodos nos encontramos con:
- **Constructor:** crea una instancia de la clase y establece el estado inicial del objeto.
- **Selector:** selecciona una parte del estado o devuelve una propiedad resultante de la combinación de algunos atributos.
- **Modificador:** modifica una parte del estado según algún criterio.
- **Visualizador:** muestra todo el estado del objeto de una forma elaborada.

Gestión de objetos en Java



Cuando una referencia no apunta a un objeto su valor es igual a null



No se puede trabajar con referencias a null

```
public static void main(String[] args){
    Deposito dp1=new Deposito(..);
    Deposito dp2; //Por defecto vale null
    dp2.agregarAgua(300);
```

```
Exception in thread "main" java.lang.NullPointerException
        at tema5.Tema5.main(Tema5.java:21)
C:\Users\Dani\AppData\Local\NetBeans\Cache\8.2\executor-sni
BUILD FAILED (total time: 2 seconds)
```

- La definición anterior de atributos y métodos no tiene ningún modificador de accesibilidad por lo que no tienen encapsulamiento.
- Los modificadores de accesibilidad son **public, private y protected**.
- **public** indica que se puede acceder desde cualquier clase.
- **private** indica que solo es accesible desde la propia clase que lo define.
- **protected** es igual que private pero también es accesible desde una que herede de ella.

- El principal mecanismo de encapsulamiento es definir la accesibilidad de los atributos y la interfaz publica a través de los métodos de una clase.
- La interfaz es lo que permite hacer la clase con la información que almacena un objeto en concreto.
- Los métodos son una membrana exterior que esconden los detalles de implementación al usuario de la clase.
- La encapsulación hace más fácil de comprender y mantener un sistema.
- En el ejemplo del deposito no podemos cumplir el contrato del objeto que vimos anteriormente.

Sin control de accesibilidad

```
//Deja al deposito en un estado incongruente
dp.id_deposito="Codigo rarororaro";
dp.volumen_actual=-230;
dp.capacidad=500;
dp.volumen_actual=600;

System.out.println("============================");
System.out.println("|ID Deposito: "+dp.id_deposito+"m3    |");
System.out.println("|Capacidad total: "+dp.capacidad+"m3 |");
System.out.println("|Volumen actual: "+dp.volumen_actual+"m3  |");
System.out.println("============================");
```

Con control de accesibilidad

```
public class Depositos {
    private double capacidad;
    private double volumen_actual;
    private String id_deposito;
```

```
id_deposito has private access in Depositos
---
(Alt-Enter muestra sugerencias)
        dp.id_deposito="Codigo rarororaro";
        dp.volumen_actual=-230;
        dp.capacidad=500;
        dp.volumen_actual=600;

        System.out.println("============================");
        System.out.println("|ID Deposito: "+dp.id_deposito+"m3    |");
        System.out.println("|Capacidad total: "+dp.capacidad+"m3 |");
        System.out.println("|Volumen actual: "+dp.volumen_actual+"m3   |");
        System.out.println("============================");
```

- Para entender mejor todo esto pensemos en una clase que define una cuenta corriente.

```
public class CuentaCorriente {
    String nombre;
    String codigo_cuenta;
    String direccion;
    double saldo;
```

- Al tener acceso libre a los atributos podemos asignarle el valor que queramos, esto puede parecernos correcto pero elimina las ventajas de la programación orientación a objetos, convirtiendo las clases en meros contenedores de datos sin ningún control.

La clase no podría controlar:

- Asignación de saldos negativos.
- Limite de ingreso o de retirada.
- El número de cuenta es único y tiene una forma concreto.
- Si se añade más información a la clase, el programador que la use tiene modificar su código para imprimir dicha información.

Cuenta bien definida y contrato del objeto

```java
public class CuentaCorriente {
    private String nombre;
    private String codigo_cuenta;
    private String direccion;
    private double saldo;
```

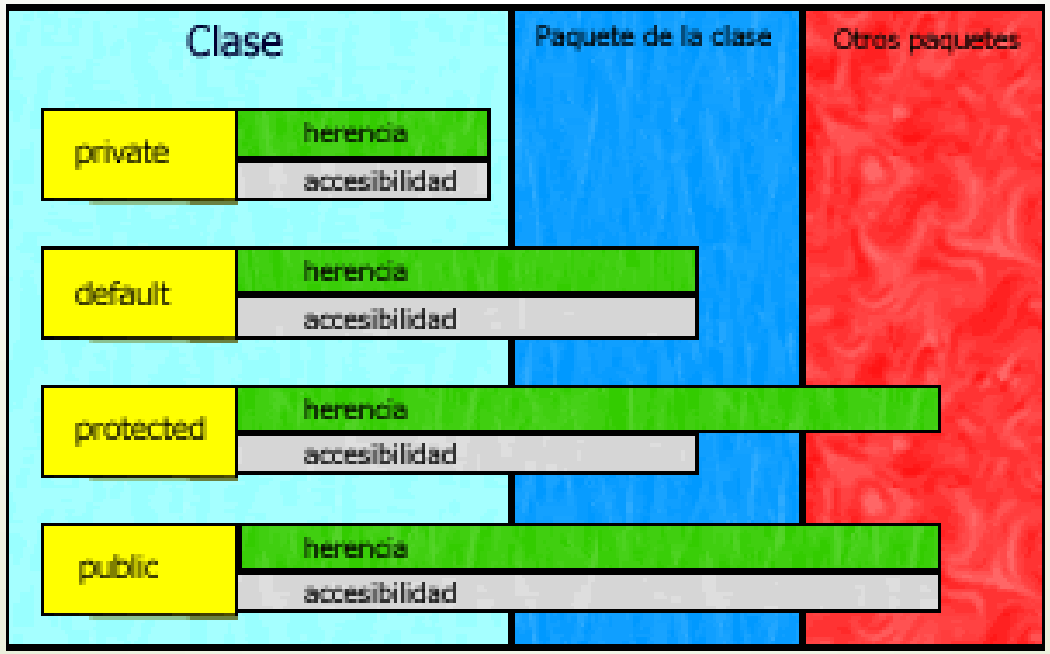
```java
//Modificadores
public void modificarDireccion(String nueva){
    direccion=nueva;
}
public void ingresar(double cantidad){
    if(cantidad<=500){
        saldo+=cantidad;
    }else{
        System.out.println("ERROR: No puedes ingresar mas de 500");
    }
}
public void retirar(double cantidad){
    if(cantidad<=saldo){
        if(cantidad<=300){
            saldo-=cantidad;
        }else{
            System.out.println("ERROR: Superar el limite para retirar");
        }
    }else{
        System.out.println("ERROR: No tienes dinero suficiente");
    }
}
```

En resumen:

- El encapsulamiento permite que el objeto no se pueda encontrar en **un estado erróneo** ya que la única manera de acceder a los atributos del objeto es a través de los métodos (interfaz).
- Por ejemplo, una cuenta corriente no puede tener números rojos.
- Los programadores que usen una clase si solo usan la interfaz se aseguran que, si se modifica la implementación de la clase **no tienen que modificar su propio programa**.

Unas pautas a la hora de definir la accesibilidad son:

- Los atributos deben ser privados
- Los métodos que definen el comportamiento de la clase (interfaz) son públicos.
- Si se piensa crear una clase de la que posteriormente se herede se debe poner protected.



- En la clase Cuenta podemos observar que existen definidos más de un constructor.

```
//Constructores
public CuentaCorriente(String n,String cc,String dir){
    nombre=n;
    codigo_cuenta=cc;
    direccion=dir;
    saldo=0;
}

public CuentaCorriente(String n,String cc,String dir,double saldo_inicial){
    nombre=n;
    codigo_cuenta=cc;
    direccion=dir;
    saldo=saldo_inicial;
}
```

- Esto es posible porque en los lenguajes OO existen el concepto de **sobrecarga** que permite definir varias veces el mismo método con distintos parámetros.

- Dentro del contexto de un objeto no es necesario que se especifique a qué objeto hace referencia.
- Sin embargo, hay casos en los que puede ser interesante disponer de una **referencia al propio objeto** que sea crea.
- Por ejemplo, para distinguirlo de una variable o un parámetro homónimo.
- Para hacer lo comentado anteriormente utilizamos la palabra **this** que hace referencia al propio objeto.
- Veamos la clase Deposito y Cuenta con ese cambio.

Uso del this clase deposito

```
public Depositos(String id_deposito,double capacidad){
    this.capacidad=capacidad;
    this.id_deposito=id_deposito;
    this.volumen_actual=0;
}
public double obtenerVolumenActual(){
    return this.volumen_actual;
}
public void agregarAgua(double cantidad){
    if((this.volumen_actual+cantidad)<=this.capacidad){
        this.volumen_actual+=cantidad;
    }else{
        System.out.println("STOP: Posible desbordamiento");
    }
}
public void info_deposito(){
    System.out.println("==========================");
    System.out.println("|ID Deposito: "+this.id_deposito+"m3    |");
    System.out.println("|Capacidad total: "+this.capacidad+"m3 |");
    System.out.println("|Volumen actual: "+this.volumen_actual+"m3   |");
    System.out.println("==========================");
}
```

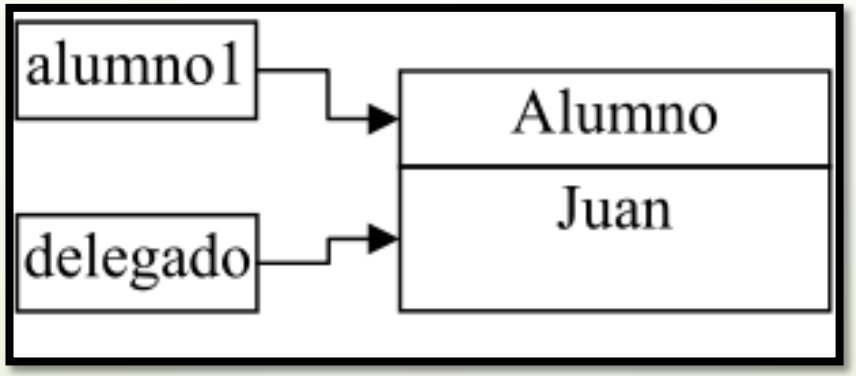
Uso del this clase Cuenta

```
public CuentaCorriente(String nombre,String codigo_cuenta,String direccion,d
    this.nombre=nombre;
    this.codigo_cuenta=codigo_cuenta;
    this.direccion=direccion;
    this.saldo=saldo_inicial;
}
public double obtenerSaldo(){
    return this.saldo;
}
public void ingresar(double cantidad){
    if(cantidad<=500){
        this.saldo+=cantidad;
    }else{
        System.out.println("ERROR: No puedes ingresar mas de 500");
    }
}
```

- Dentro de un programa orientado a objetos existen una serie de métodos utilitarios para su uso en ciertas situaciones claves y que tienen una definiciones particulares en Java, siendo:
  - Copiar objetos (constructor de copias).
  - Comparación de igualdad (equals).
  - Convertir en cadena (toString).

- El constructor de copias es un método constructor necesario para clonar objetos, porque recordamos que en Java la asignación de objeto lo que hace es apuntar al mismo objeto no crea uno nuevo.

```
Alumno alumno1=new Alumno(....);
Alumno delegado;

delegado=alumno1;
```



Constructor de copias en CuentaCorriente

```
public class CuentaCorriente {
    private String nombre;
    private String codigo_cuenta;
    private String direccion;
    private double saldo;
```

```
public class CuentaCorriente{
    ...
    public CuentaCorriente(CuentaCorriente c){
        this.nombre=c.nombre;
        this.codigo_cuenta=c.codigo_cuenta;
        this.direccion=c.direccion;
        this.saldo=c.saldo;
    }
    ...
    public static void(String[] args){
        CuentaCorriente ejemplo=new CuentaCorriente("Juan","1231231","Micasa",122.98);
        CuentaCorriente copia=new CuentaCorriente(ejemplo);
    }
}
```

- Algo parecido pasa con la comparación de igualdad ya que el operador == solo compara que las variables apunten al mismo objeto.
- Esta es la misma situación comparando String que usamos el método equals.
- De hecho en Java si queremos definir cuando dos objetos son iguales hay que definir el método equals es un estándar.
- Para definir cuando dos objetos son iguales no necesariamente tiene que tener todos sus campos iguales sino que solo si son iguales campo o campos claves

Método equals

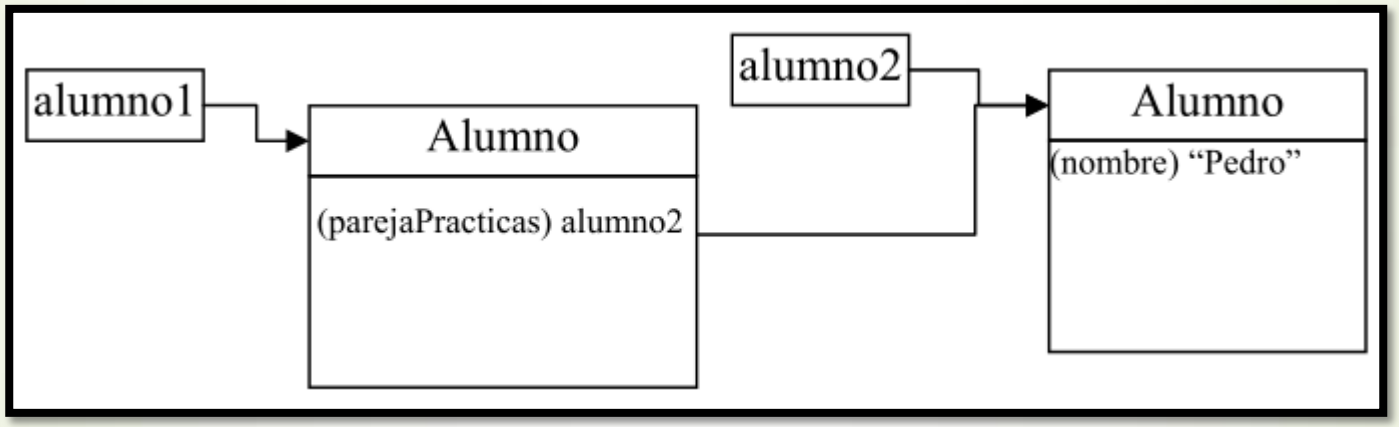
```
public class Rectangulo{
    private double alto;
    private double ancho;
    ...
    public boolean equals(Rectangulo r){
        return this.alto==r.alto && this.ancho==r.ancho;
    }
}

public class Persona{
    private String nombre;
    private String DNI;
    ...
    public boolean equals(Persona p){
        return this.DNI.equals(p.DNI);
    }
```

- El método visualizador en lugar de ser void y mostrar datos por pantalla debe devolver un String.
- Esto es así para hacerlo independiente del dispositivo de salida en el que se vaya a mostrar.
- En Java existe un método definido en cualquier clase para ese cometido llamado **toString**.

Método toString clase Deposito

```
public class Deposito{
    ...
    public String toString(){
        String res;
        res="========================="+
            "ID Deposito:"+this.id_deposito+"m3\n"+
            "Capacidad total:"+this.capacidad+"m3\n"+
            "Volumen actual:"+this.volumen_actual+"m3\n"+
            "=========================";
        return res;
    }
}
```

- Un atributo de una clase puede ser otra clase.
- Se denomina composición de clases y es el primer mecanismo básico del desarrollo orientado a objetos junto con la herencia.



- Dentro de la composición veremos un concepto vital que es la delegación de clases, donde una clase llama a un método de otra clase para realizar una tarea.

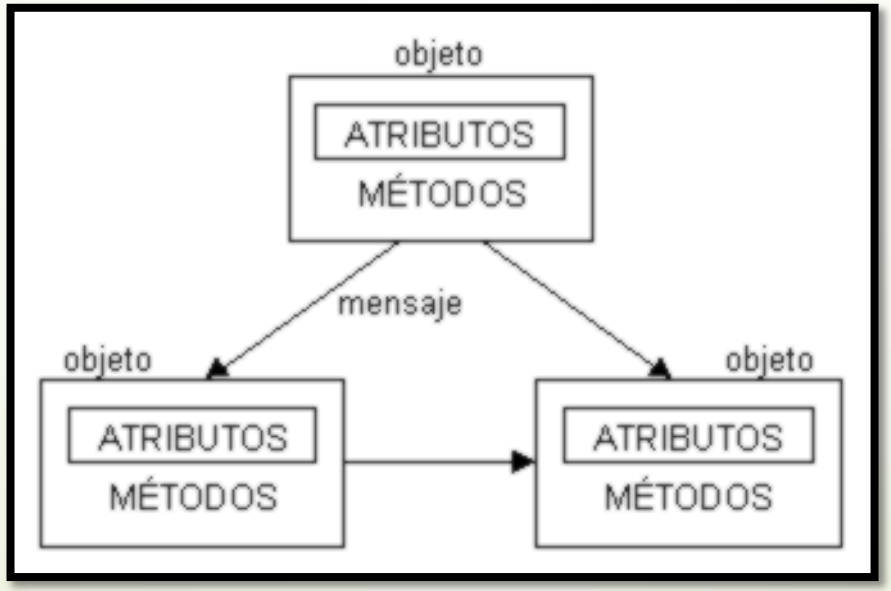
Clase Punto

```java
public class Punto {
    private double x,y;

    public Punto(double x,double y){
        this.x=x;
        this.y=y;
    }

    //Calcula la distancia entre este punto y otro
    public double distanciaPunto(Punto p){
        double distX=this.x-p.getX();
        double distY=this.y-p.getY();

        return Math.sqrt(distX*distX-distY*distY);
    }

    public String toString(){
        String res="============================\n"
                  +"|Coordenada X: "+this.x+"metros |\n"
                  +"|Coordenada Y: "+this.y+"metros |\n"
                  +"============================\n";

        return res;
    }
```

Clase Linea

```java
public class Linea {
    private Punto inicio,fin;

    public Linea(double x1,double y1,double x2,double y2){
        this.inicio=new Punto(x1,y1);
        this.fin=new Punto(x2,y2);
    }

    public double longitudLinea(){
        return inicio.distanciaPunto(fin);
    }

    public void desplazarFin(double x,double y)
    {
        this.fin.desplazar(x, y);
    }

    public String toString(){
        String res="Punto inicial:\n"+this.inicio.toString()+"\n"
                  +"Punto final:\n"+this.fin.toString();
    }
```

- En este ejemplo la delegación consiste en que la clase Linea se compone de la clase Punto y para completar métodos de la clase Linea esta delega en la clase Punto
- En concreto cuando un objeto de una clase llama otro en una delegación dicha llamada se llama mensaje.



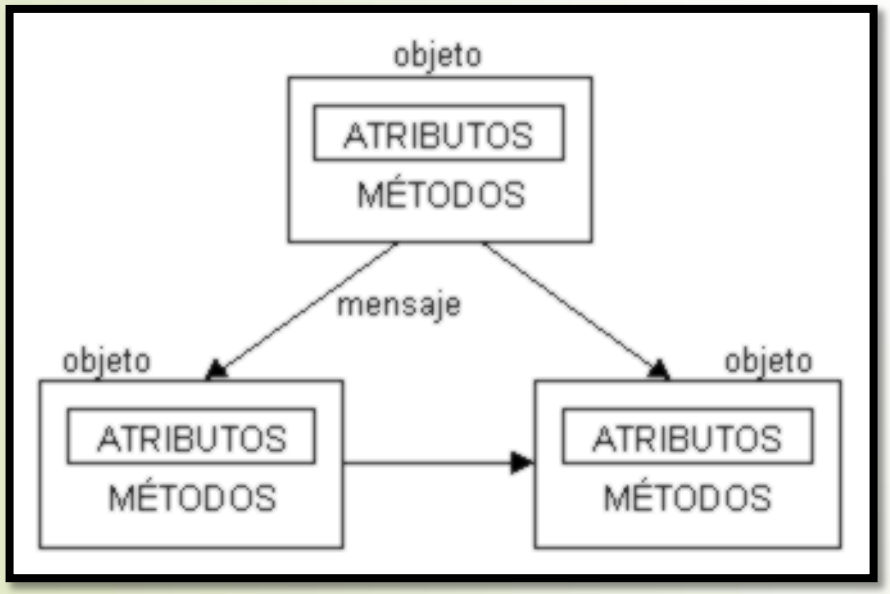
Clase Motor

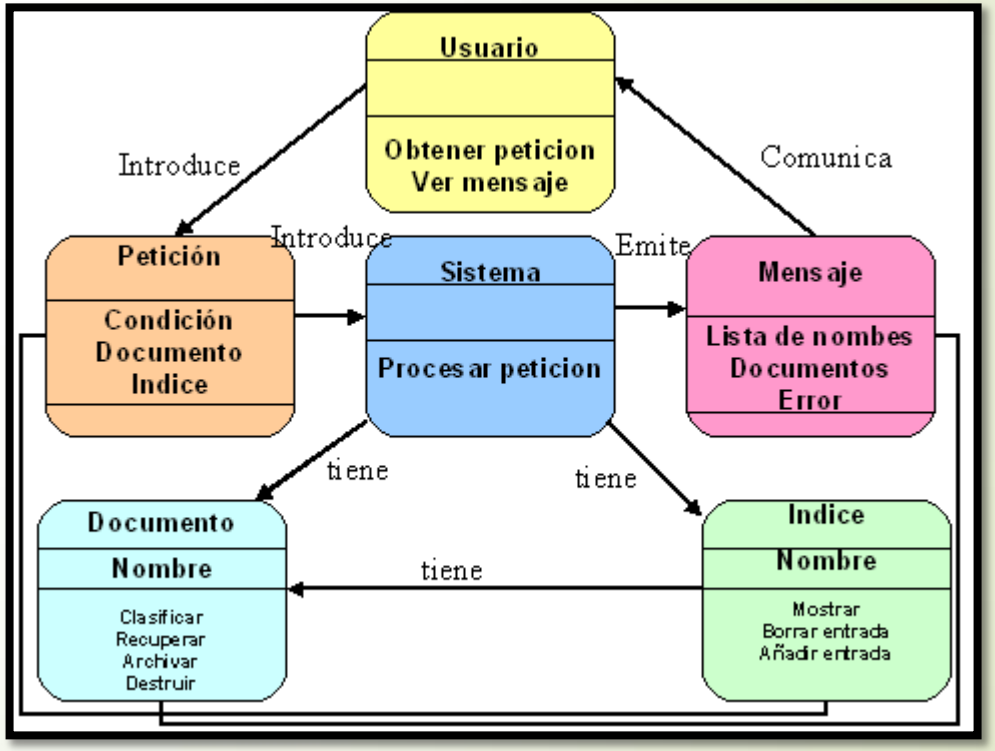
```
public class Motor{
    private int potencia;
    private int consumo;
    private int velocidad_max;

    public Motor(int pot, int con, int vel){…}
    public int getPotencia(){…}
    public int getConsumo(){ … }
    public int getVelocidad(){…}
    public void arrancar(){…}
    public void apagar(){…}
    public void acelerar(){…}
    public void info_motor(){…}
}
```

Clase Coche

```
public class Coche{
    private int bastidor;
    private Motor unmotor;
    …
    public Coche(int bastidor, int potencia, int consumo, int velocidad){
        this.bastidor=bastidor;
        unmotor= new motor(potencia, consumo, velocidad);
    }
    public void arrancar(){
        unmotor.arrancar();
        …
    }
    public void acelerar(){
        unmotor.acelerar();
        …
    }
    …
    public void resumen_coche(){
        System.out.println("Numero de bastidor: "+this.bastidor);
        unmotor.info_motor();
    }
}
```

- En un diseño orientado a objetos todo el software se divide en objetos y se encadenan llamadas a métodos para que el programa realice su cometido.
- De forma parecida al diseño de base de datos E-R





Variables estáticas (static)

- Cuando declaramos una variable con el modificador static indicamos que dicha variable estará disponible sin necesidad de crear una instancia.
- Además dicha variable será única para todas las instancias de la clase, dicho de otro modo será una variable compartida.
- Si además le acompañamos de la palabra **final** será una variable que no se podrá modificar, en otras palabras una constante.
- También se les llama **variables de clase**.

- Por ejemplo en la clase Deposito queremos que cada deposito tenga un identificador distinto.
- Para ello tendríamos que programar nosotros una búsqueda en todos las instancias que haya creadas si ya esta cogido un id que generemos nosotros.
- Eso puede tener un coste de procesamiento muy alto.
- Usando variables static se resuelve fácilmente.
- Pensemos en que el deposito va a tener un prefijo y eso no va a cambiar nunca y después un número que empiece por 100 y vaya aumentando conforme se vayan creando depósitos.

Uso de static en la clase Deposito

```java
public class Deposito{
    private static int siguiente_id=1;
    private final static String prefijo="DEPO-";

    private String id_deposito;
    private double capacidad;
    private double volumen_actual;

    public Deposito(double capacidad){
        //Creamos el id del nuevo deposito
        this.id_deposito=this.prefijo+this.siguiente_id;
        //Actualizamos para el siguiente deposito
        //que se vaya a crear
        this.siguiente_id++;
        //Inicializar resto de variables
        this.volumen_actual=0;
        this.capacidad=capacidad;
    }
```

Bibliografía

- **García de Jalón, j.: "Aprende Java como si estuvieras en primero".** Editorial TECNUN. 2000
- **Holzner, S.: "La biblia de JAVA 2".** Editorial Anaya Multimedia 2000.
- **Moreno Pérez, J.C.: "C.F.G.S Entornos de desarrollo"** Editorial RA-MA. 2012
- **Wikipedia, la enciclopedia libre. http://es.wikipedia.org/** Última visita: Octubre 2018.
- **López, J.C.: "Curso de JAVA http://www.cursodejava.com.mx** Última visita: Octubre 2018**.**
- **Documentación oficial Java JSE 8 http://docs.oracle.com/javase/8/** Última visita: Octubre 2015.
- **Programación en castellano: Java. http://www.programacion.net/java Última visita: Octubre 2015.**