

Programación Orientada a Objetos

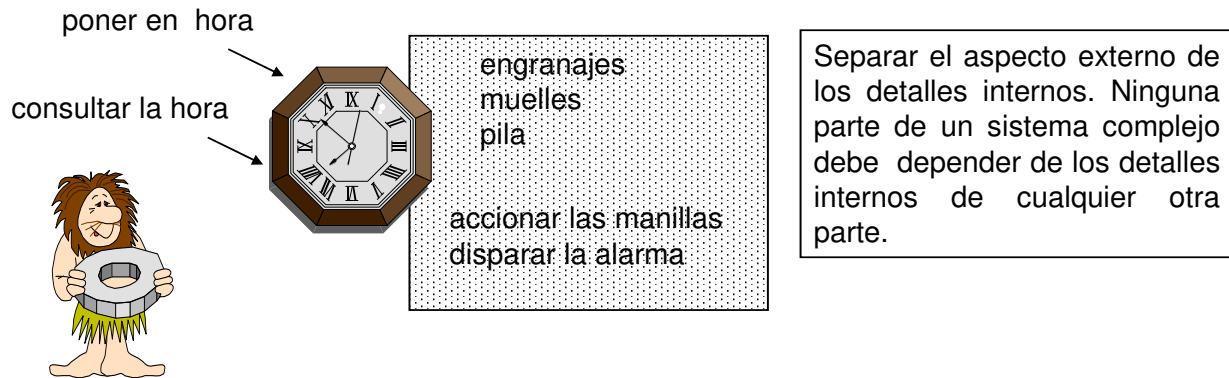
Tema 2: Fundamentos de la programación orientada a objetos

Tema 2-5: Conceptos avanzados de OO

- Tema 2-5: Conceptos avanzados de OO
- 1. Encapsulación y Ocultación de información
- 2. Cohesión y acoplamiento
- 3. Comunicación
- 4. Polimorfismo
- 5. Clases abstractas
- 6. Interfaces
- 7. Lenguajes OO
- 8. Análisis y diseño OO



- Técnica que permite localizar y ocultar los detalles de un objeto respecto al usuario, aislándolo del aspecto interno.
- Previene que un objeto sea manipulado por operaciones distintas a las que le son propias, así como que sus operaciones manipulen datos ajenos a ellas.



- Encapsulación = disimulación de los datos.
- Los datos están protegidos de los objetos exteriores.
- Los objetos son independientes de la estructura interna de otros objetos.
- Es la propiedad que asegura que la información de un módulo esta oculta al mundo exterior.
- Es una técnica de diseño para descomponer sistemas en módulos.
- Permite la constitución de bibliotecas de objetos genéricos reutilizables.

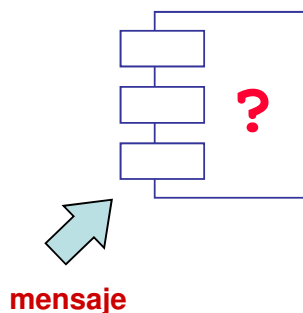


- **Programación estructurada:**

- Funciones a realizar.
- Variables, constantes.
- El programador concentra su atención sobre el desarrollo de rutinas. La utilización de nuevas referencias a estas rutinas no se tiene en cuenta por el código existente.
- Los costes de mantenimiento aumentan.

- **Programación orientada a objetos:**

- Responsabilidades a atribuir.
- Clase, atributos, métodos.
- El desarrollador segmenta su aplicación en componentes materializados en clases.
- Los datos están protegidos del exterior (el acceso a los atributos sólo puede hacerse a través de los métodos).



**El módulo es
una “caja negra”**

- Principio por el cual los módulos son utilizados por su **especificación**, no por su implementación.

- Los detalles de implementación están ocultos y sólo es accesible la **interfaz**.

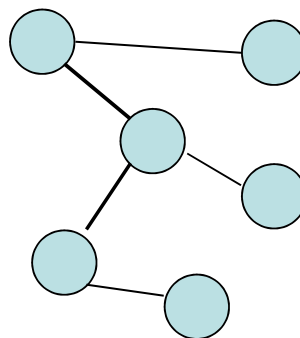
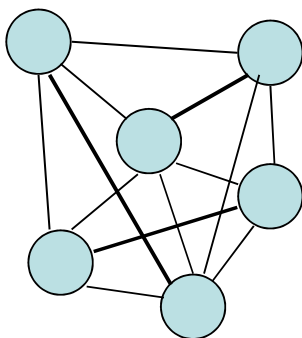
- Un objeto tiene una **interfaz pública** que otros objetos pueden usar para comunicarse con él. El objeto puede mantener información privada y métodos que pueden cambiar sin que esto afecte a otros objetos que dependen de él.



Buscamos: *MAXIMA cohesión / MINIMO acoplamiento*

Acoplamiento: Es una medida de la **fuerza de conexión entre dos componentes de un sistema**.
Cuanto más **conoce** un componente de otro, más fuerte es el acoplamiento entre ellos.

- Cuando construimos sistemas de objetos, debemos acoplar los objetos en cierta medida.
- Si al diseñar un objeto lo dotamos de un **excesivo conocimiento de otros objetos**, estamos haciendo acoplamiento innecesario.



En un sistema **fuertemente acoplado**:

- Disminuye la posibilidad de **reutilizar objetos** individuales.
- El sistema es más complejo y por tanto **más difícil de entender y mantener**.



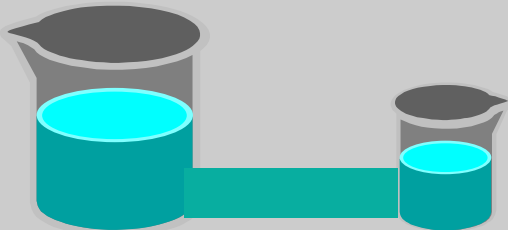
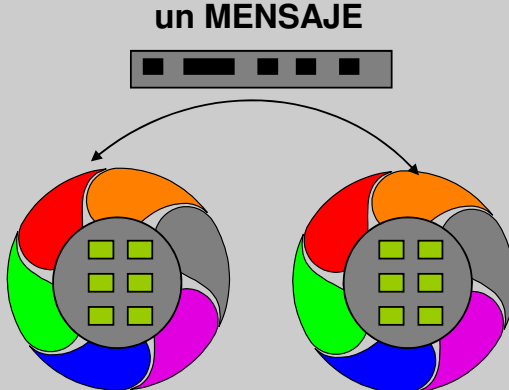
Cohesión: Es una medida de **cuánto están relacionados lógicamente** los componentes encapsulados en un objeto.

Es la otra cara de la moneda del acoplamiento.

Para un objeto **poco cohesionado**:

- Aumenta la probabilidad de **sufrir cambios**.
- Disminuye la probabilidad de **reutilización**.



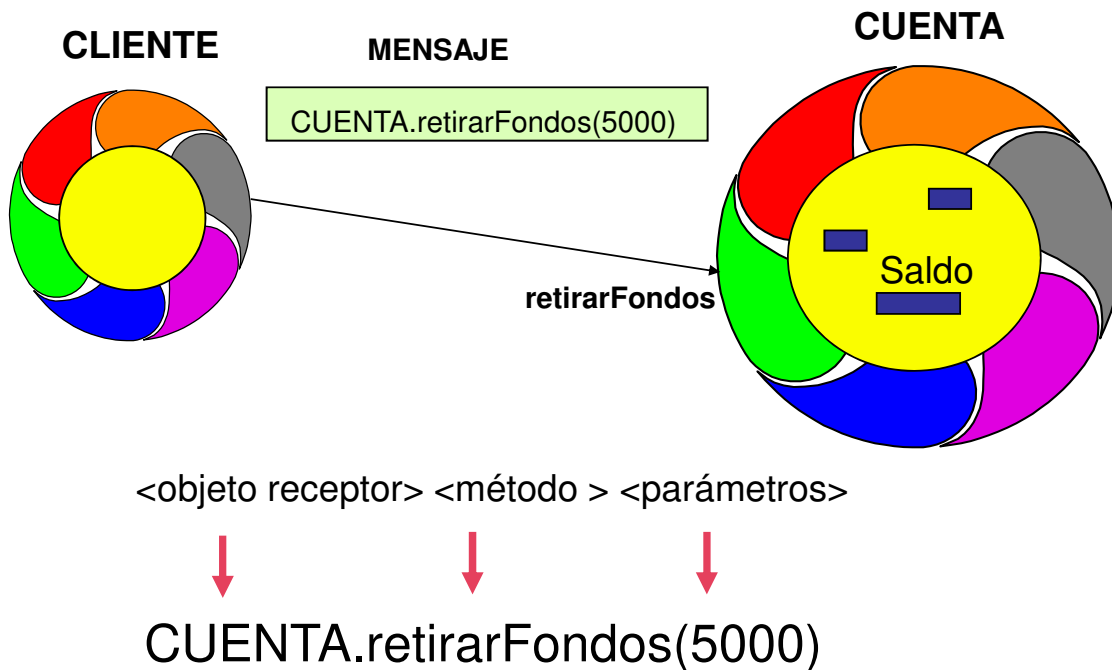
MUNDO REAL	SIMULACIÓN INFORMÁTICA
<p>Dos elementos de un sistema físico interactúan intercambiando: FLUJOS</p> <p>de datos, materia, o energía.</p> 	<p>Dos objetos interactúan intercambiando: MENSAJES</p> <p>un MENSAJE</p> 



- Los objetos se comunican entre sí mediante el envío de mensajes.
- El propósito de un mensaje es pedir al objeto que lo recibe que active el método que se le indica y que devuelva al objeto original el resultado de esa acción.
- Una secuencia de instrucciones en un lenguaje clásico es reemplazada por una secuencia de comunicaciones.
- La sintaxis de una comunicación es muy simple:
 - <sujeito> <verbo> <complemento>
 - o también:
 - <destinatario> <mensaje> <datos>
 - <objeto receptor> <método seleccionado> <parámetros>



- Los mensajes que recibe un objeto son los únicos conductos que conectan el objeto con el mundo exterior.
- Los datos de un objeto están disponibles para ser manipulados solo por los métodos del propio objeto.
- Cuando se ejecuta un POO ocurren tres cosas:
 1. Los objetos se crean a medida que se necesitan.
 2. Los mensajes se mueven de un objeto a otro (o desde el usuario a un objeto) a medida que el programa procesa la información.
 3. Cuando los objetos no se necesitan se borran y se libera la memoria.



- **Unificación de la sintaxis**

- funciones:
 - restar (a, b) : restar a de b o restar b de a ?
- expresiones (notación postfija, prefija, infija):
 - a b - , - a b , a - b
- mensaje: a restar: b, a - b

- **Simplificación de la sintaxis**

- "El gato come al ratón"
- Qué es lo más natural ?
 - Comer (gato, ratón)
 - gato.come(ratón)



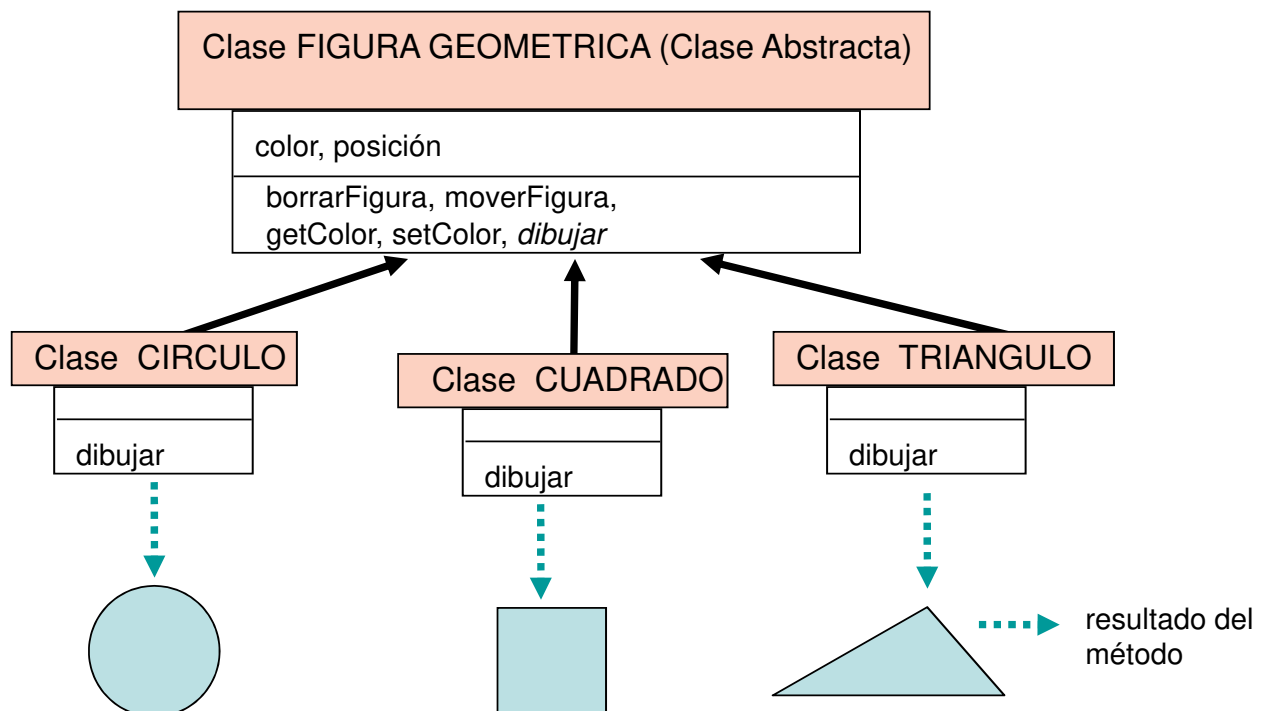
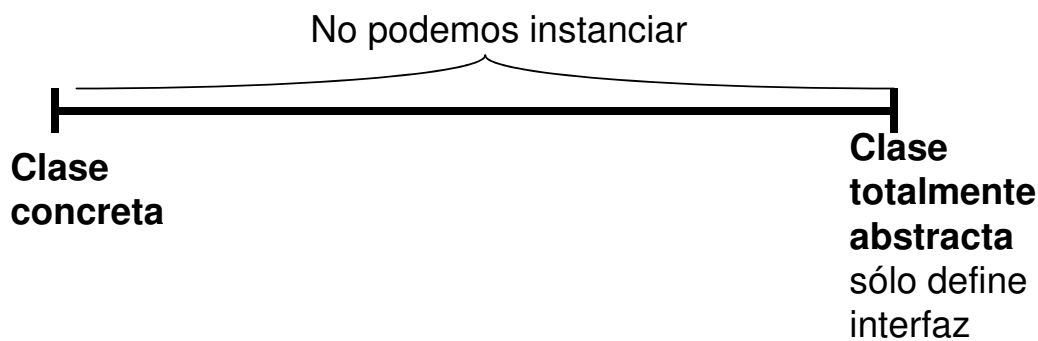
- Significa ***tener o asumir distintas formas***.
- En el contexto de POO se refiere a la **capacidad de los diferentes objetos para responder de distinta forma a la misma operación**. Esta característica habilita al programador para tratar uniformemente objetos que provienen de clases diferentes.
- Permite enviar **el mismo mensaje a objetos diferentes** y que cada uno responda en la forma apropiada según el tipo de objeto que sea, ejecutando **su método**.



- El **polimorfismo** está asociado a la **ligadura dinámica** (*“dynamic binding”*). La asociación de un **método** con su nombre no se determina hasta el momento de su ejecución.
- El **polimorfismo** es una aplicación particular de la **herencia** de comportamiento. Una misma operación (un mismo elemento de comportamiento) podrá interpretarse de diversas maneras según la posición en la que se encuentre dentro de la jerarquía.
- Usualmente requiere del empleo de **clases abstractas**.

•Clases Abstractas:

- A veces es útil aislar en clases conceptos **incompletos** aunque coherentes y cohesivos.
- Los atributos y los métodos de estas clases están disponibles para ser especializados vía herencia.
- Una clase abstracta no puede tener instancias.





- **Ventajas:**
 - Da uniformidad a la sintaxis.
 - Disminuye la cantidad del código a escribir (por eliminación de las estructuras alternativas con opciones múltiples).
 - Facilita el tratamiento de colecciones heterogéneas.



- Cuando se organizan las clases en una jerarquía lo normal es que las clases que representan los conceptos más abstractos ocupen un lugar más alto en la misma.
- Los lenguajes orientados a objetos dan la posibilidad de declarar clases que definen como se utiliza pero sin tener que implementar los métodos que posee.
- En Java se realiza mediante clases abstractas que poseen métodos abstractos que implementarán las clases hijas.



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

public abstract class FiguraGeometrica {
    private Punto posicion;
    private String color;
    ...
    public void mover(Punto nuevaPos) {
        posicion = nuevaPos;
        dibujar();
    }
    public abstract void dibujar();
}

public final class Circulo extends FiguraGeometrica {
    @Override
    public void dibujar() {
        ...
    }
}
```

No se pueden crear instancias de las clases abstractas. Tienen la función de encapsular un concepto abstracto que compartirán sus subclases.

Parte de la interfaz que definen, no posee una implementación.

No se puede extender (final). Suelen utilizarse cuando pueda ser "peligroso" permitir que las subclases dieran otra implementación de ciertos métodos.



- Constituyen un mecanismo para separar la interfaz de la implementación.
- Una interface contiene métodos y atributos constantes. Las clases que implementen la interface están obligadas a codificar todos los métodos definidos en la interface.
- Se declaran de forma similar a las clases:

```
interface Nombre {
    atributos
    métodos
}
```

← Cuerpo de la interfaz.
Sólo puede contener signatures de métodos (no la implementación o cuerpo) y atributos

- Los atributos de una interfaz son **implícitamente public, static y final**, y deben ser inicializados en la declaración.
- Los métodos de una interfaz son **implícitamente public, abstract y no pueden ser static**.



• Ejemplo:

```
public interface VideoClip {
    int VERSION = 1; //atributo constante
    void play(); //comienza la reproducción del video
    void bucle(); //reproduce el video en un bucle
    void stop(); //detiene la reproducción
}

public class DivX implements VideoClip {
    public void play() {
        <código>
    }
    public void bucle() {
        <código>
    }
    public void stop() {
        <código>
    }
}
```



Lenguajes Orientados a Objetos



- **Lenguajes de programación OO puros:** Solo soportan el paradigma de programación OO. Todo son objetos, clases y métodos. Ej: Java.
- **Lenguajes de programación OO híbridos:** Soporta otros paradigmas además del OO. Ej: C++.

/******

- **Tipificación:** proceso de declarar cuál es el tipo de información que contiene una variable.

- **Tipificación fuerte, estricta o estática.** En tiempo de compilación.

- **Tipificación débil, no estricta o dinámica.** En tiempo de ejecución.

/******

- **Ligadura:** Proceso de asociar un atributo a un nombre, o la llamada a una función con su código real.

- **Ligadura estática, temprana o anticipada:** En tiempo de compilación.

- **Ligadura dinámica, retardada o postergada:** En tiempo de ejecución. LOO. Permite el polimorfismo.

- Desarrollo de software OO:
 - Encontrar los objetos relevantes.
 - Describir los tipos de objetos.
 - Encontrar las operaciones para los tipos de objetos.
 - Encontrar relaciones entre objetos.
 - Utilizar los tipos de objetos y las relaciones para estructurar el software.

