Tema 1 (Repaso)

- Objeto: Entidad que tiene identidad propia. Són diferenciables entre ellos.
  - Avión con matrícula 327
  - La factura 3443
  - Avión con matrícula 999
  - Un semáforo concreto
  - Una manzana

Tema 1 (Repaso)

- Clase: describe un conjunto de objetos con:
  - Las mismas propiedades
  - Comportamiento común
  - Idéntica relación con otros objetos
  - Resultado del proceso de **abstracción**: eliminar diferencias entre objetos para poder observar aspectos comunes
    - Avión 329, Avión 999 ⟶ clase Avión
  - Los objetos de una clase tienen las mismas propiedades y los mismos patrones de comportamiento.

Tema 1 (Repaso)

- Atributo: propiedad compartida por los objetos de una clase.
  - Ejemplos:
    - Persona => Nombre, dirección, teléfono, …
    - Avión => modelo, capacidad, color, …
  - Cada atributo tiene un valor (probablemente diferente) por cada instancia de un objeto.
  - Pueden ser básicos o derivados.

Tema 1 (Repaso)

- Método: Implementación de una función o transformación que se puede aplicar a los objetos de una clase
  - Los métodos son invocados por otros objetos a través de un **mensaje**
  - Hay que indicar el tipo de los argumentos y del resultado:
    - Ejemplo:
      - Persona.tieneMismoNombre(P: Persona): booleano

Tema 2

Encapsulación, herencia y polimorfismo

Information hiding

- Ocultación de decisiones de diseño en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen.

- Proteger una decisión de diseño supone proporcionar una interfaz estable que proteja el resto del programa de la implementación (susceptible de cambios).

Information hiding (ejemplo)

```
class Vector {
    public int x;
    public int y;
    public int z;

}


public class Cliente {

    public static void main(String [] args) {
        Vector vector = new Vector();
        vector.x = 3;
        vector.y = 2;
        vector.z = 1;


        double modulo = Math.sqrt(Math.pow(vector.x, 2) + Math.pow(vector.y, 2) + Math.pow(vector.z, 2));

        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

Qué sucede si decido implementar los puntos del vector con un int[3] ???

Information hiding (ejemplo)

```
class Vector {
    public int[] points = new int[3];
}


public class Cliente {

    public static void main(String [] args) {
        Vector vector = new Vector();
        vector.points[0] = 3;
        vector.points[1] = 2;
        vector.points[2] = 1;

        double modulo = Math.sqrt(Math.pow(vector.points[0], 2) + Math.pow(vector.points[1], 2) + Math.pow(vector.points[2], 2));

        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

Hay que cambiar 3 sitios del código. Os imaginais un programa grande que tuviera que calcular en varios sitios el módulo del vector?

Information hiding

Definición: ***La técnica de esconder detalles de implementación al usuario se llama Information Hiding***

Todos los atributos escondidos dentro de la clase

- NO usar propiedades públicas (+*Name*)
- ponerlas privadas

acceder a esas a través de métodos *getter* y *setter* públicos *(+getName, +setName).* Dichos métodos se llaman ***accessors.***

En general: todo tipo de acceso a datos (incluso a constantes) mediante métodos

Information hiding

```
class Vector {
    private int[] points = new int[3];

    public void setX(int x) {
        points[0] = x;
    }

    public void setY(int y) {
        points[1] = y;
    }

    public void setZ(int z) {
        points[2] = z;
    }

    public int getX() {
        return points[0];
    }

    public int getY() {
        return points[1];
    }

    public int getZ() {
        return points[2];
    }
}
```

```
public class Cliente {


    public static void main(String [] args) {
        Vector vector = new Vector();
        vector.setX(3);
        vector.setY(2);
        vector.setZ(1);



        double modulo = Math.sqrt(Math.pow(vector.getX(), 2) +
                Math.pow(vector.getY(), 2)
                + Math.pow(vector.getZ(), 2));


        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

Information hiding

```
class Vector {
    private int x;
    private int y;
    private int z;

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setZ(int z) {
        this.z = z;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public int getZ() {
        return this.z;
    }
}
```

```
public class Cliente {


    public static void main(String [] args) {
        Vector vector = new Vector();
        vector.setX(3);
        vector.setY(2);
        vector.setZ(1);



        double modulo = Math.sqrt(Math.pow(vector.getX(), 2) +
                Math.pow(vector.getY(), 2)
                + Math.pow(vector.getZ(), 2));


        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

La clase *Cliente* no cambia!!!!

Information hiding

No poner detalles de implementación (como métodos privados) en la interfaz pública.

El *usuario* de una clase debería ver (y aprenderse) solo lo mínimo que le hace falta para usar la clase.

Todo lo demás tiene que estar en la parte privada de la clase

Esconder también métodos cuando son detalles de implementación

Information hiding

Por otra parte, nos puede interesar permitir únicamente la consulta de los valores, no la modificación.

O nos puede interesar la consulta y modificación sólo del eje X, no del Y o del Z.

Todo esto lo conseguimos solamente actuando así: atributos privados y métodos de acceso públicos.

Information hiding - Seguridad

```
public class Fraccion {
    private int numerador, denominador;


    public void setDenominador(int valor) {
        this.denominador = valor;
    }

    public int getDenominador() {
        return this.denominador;
    }

    public void mostrar() {
        System.out.println(this.numerador + " / " + this.denominador + "= ");
        System.out.println(this.numerador / this.denominador);
    }
}
```

Si el denominador vale 0, tenemos una división por 0

Information hiding - Seguridad

```
public class Fraccion {
    private int numerador, denominador;



    public void setDenominador(int valor) {
        if (valor != 0) this.denominador = valor;
        else this.denominador = 1;
    }

    public int getDenominador() {
        return this.denominador;
    }

    public void mostrar() {
        System.out.println(this.getNumerador() + " / " + this.getDenominador() + "= ");
        System.out.println(this.getNumerador() / this.getDenominador());
    }
}
```

Encapsulación

- Punto central de la reusabilidad de código
- Un paso más adelante del information hiding
  - Es la buena implementación del Information hiding
  - Tiene que ver con la distribución del código de una aplicación
- Principio “Tell, don’t ask”

Básicamente:

¿Para que preguntar para valores a otra clase cuando puedo delegar lo que tengo que hacer enviando un mensaje?

Encapsulación

- El uso de getters/setters esconde la representación de los datos (atributos), pero no esconde la implementación de los comportamientos sobre esos datos (get/set son métodos que no contienen lógica).

- La solución para esconder también la implementación de la lógica sobre esos datos es desplazar toda la lógica que actúa sobre dichos datos en la misma clase.

Encapsulación - Ejemplo

```
public class Cliente {

    public static void main(String [] args) {
        Vector vector = new Vector();
        vector.setX(3);
        vector.setY(2);
        vector.setZ(1);



        double modulo = Math.sqrt(Math.pow(vector.getX(), 2) + Math.pow(vector.getY(), 2)
                + Math.pow(vector.getZ(), 2));

        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

Qué pasa si quiero un vector en 2D???

Encapsulación - Ejemplo

```
class Vector {
    private int x;
    private int y;

    public Vector(int x, int y, int z) {
        this.x = x;
        this.y = y;
    }

    public double modulo()
    {
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
    }
}
```

Desplazamos la lógica que actúa sobre las coordenadas dentro de la clase que tiene la información, así no hay que preguntar por los valores desde otra clase.

Encapsulación - Ejemplo

```
public class Cliente {


    public static void main(String [] args) {
        Vector vector = new Vector(3, 2, 1);
        double modulo = vector.modulo();
        System.out.println("EL MODULO ES: " + modulo);
    }
}
```

Si volviéramos a necesitar un vector en 3D, no habría que tocar el código cliente en ningún sitio de la aplicación.

Encapsulación - Consecuencias

El componente es completamente independiente, y entonces reutilizable.
Contiene toda la lógica que concierne a sus datos.

Un cambio interno de lógica no afecta las llamadas y tampoco otras clases (separación entre interfaz y implementación)

Cada cambio de lógica está localizado dentro de la misma clase (no tengo porqué buscar en otras clases)

La abstracción es mejor (la clase *Vector* es más 'inteligente', tiene responsabilidad más definida)

Las clases no deberían invocar frecuentemente Getters y Setters de otras clases. La mayoría de accesos a los datos debería pasar por métodos 'inteligentes'.

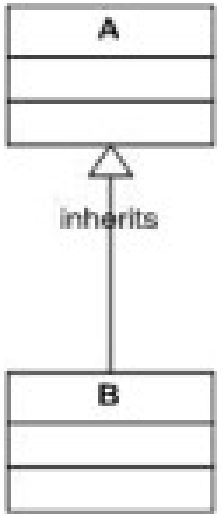
Tema 2

- Relaciones entre clases
  - Uso (un método de la clase A recibe como parámetro una instancia de B)

- Relación entre clases:
  - Contención HAS-A (la clase A tiene como propiedad una instancia de la clase B)

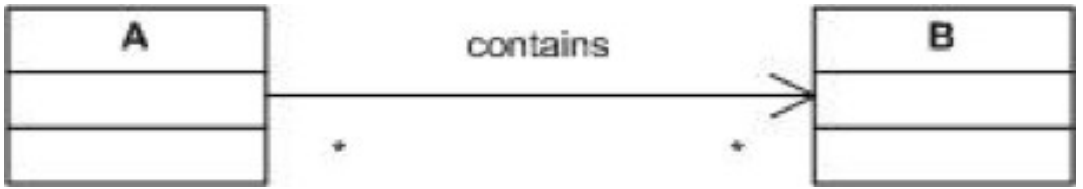
Tema 2

- Relación entre clases:
  - Herencia: la clase B hereda la estructura (propiedades y métodos) de la clase A



Tema 2

- Contención



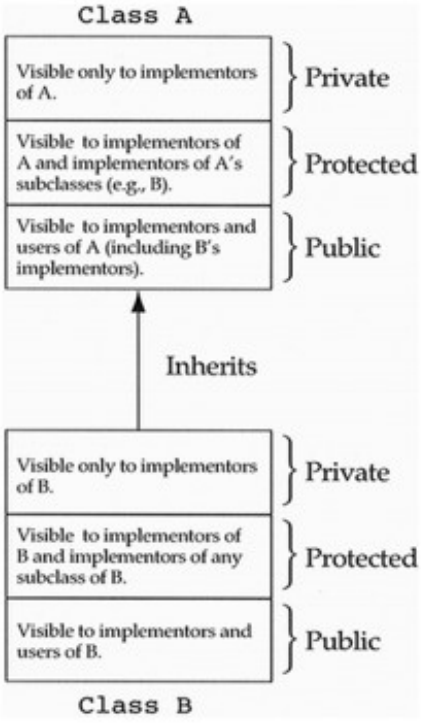
  - Ocurre si:
    - La clase A instancia a la clase B en el constructor (contención fuerte)
    - La clase A acepta una instancia de la clase B como parámetro del constructor.
    - La clase A tiene un Setter que acepta como parámetro una instanciade B

Tema 2

- Herencia
  - Es una relación entre clases
  - Clase B hereda propiedades y métodos de la clase A
    - Todas las instancias de B tendrán la misma estructura de la clase A, más sus propios miembros.
    - => todas las instancias de la clase B podrán ser utilizadas como si fueran de la clase A
    - A se llama clase base y B se llama clase derivada
    - Introduce un nuevo nivel de Information Hiding:
      - **Protected:** solo visible para las clases derivadas y la propia clase base.

Tema 2

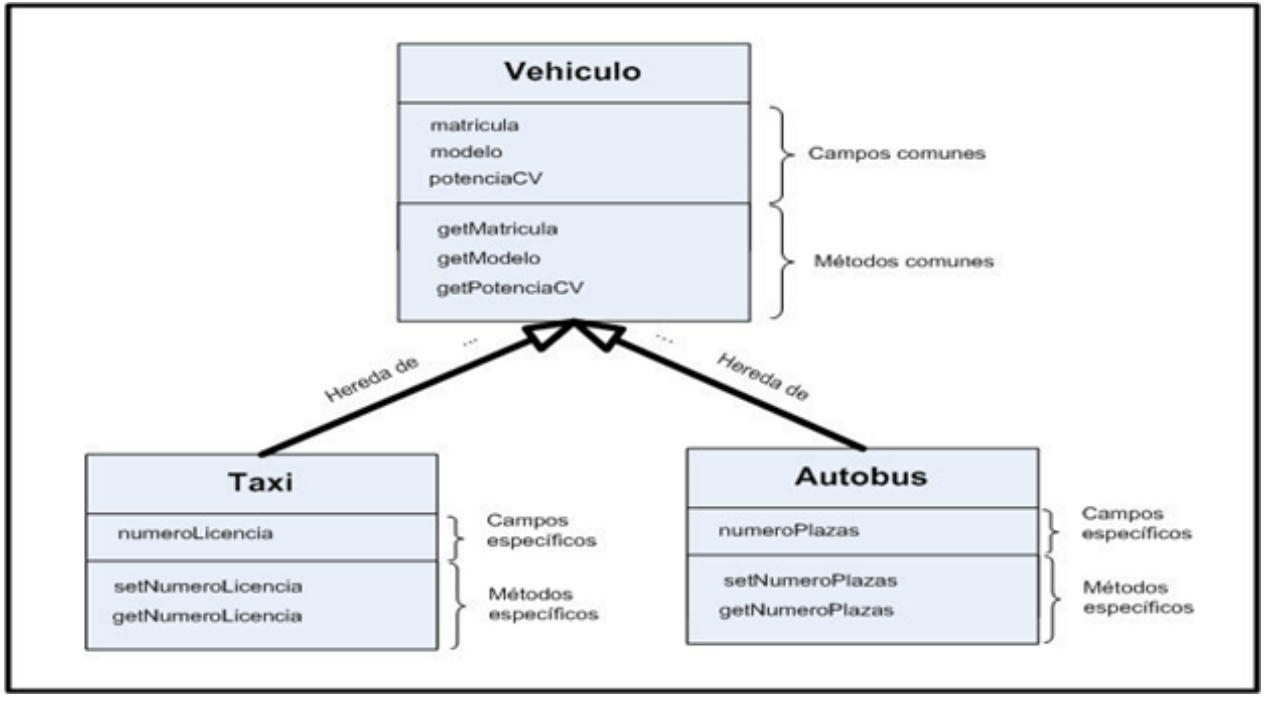
- Por las mismas razones del data hiding, es mejor dejar las propiedades de la clase base como privadas y añadir métodos de acceso protegidos para las clases derivadas



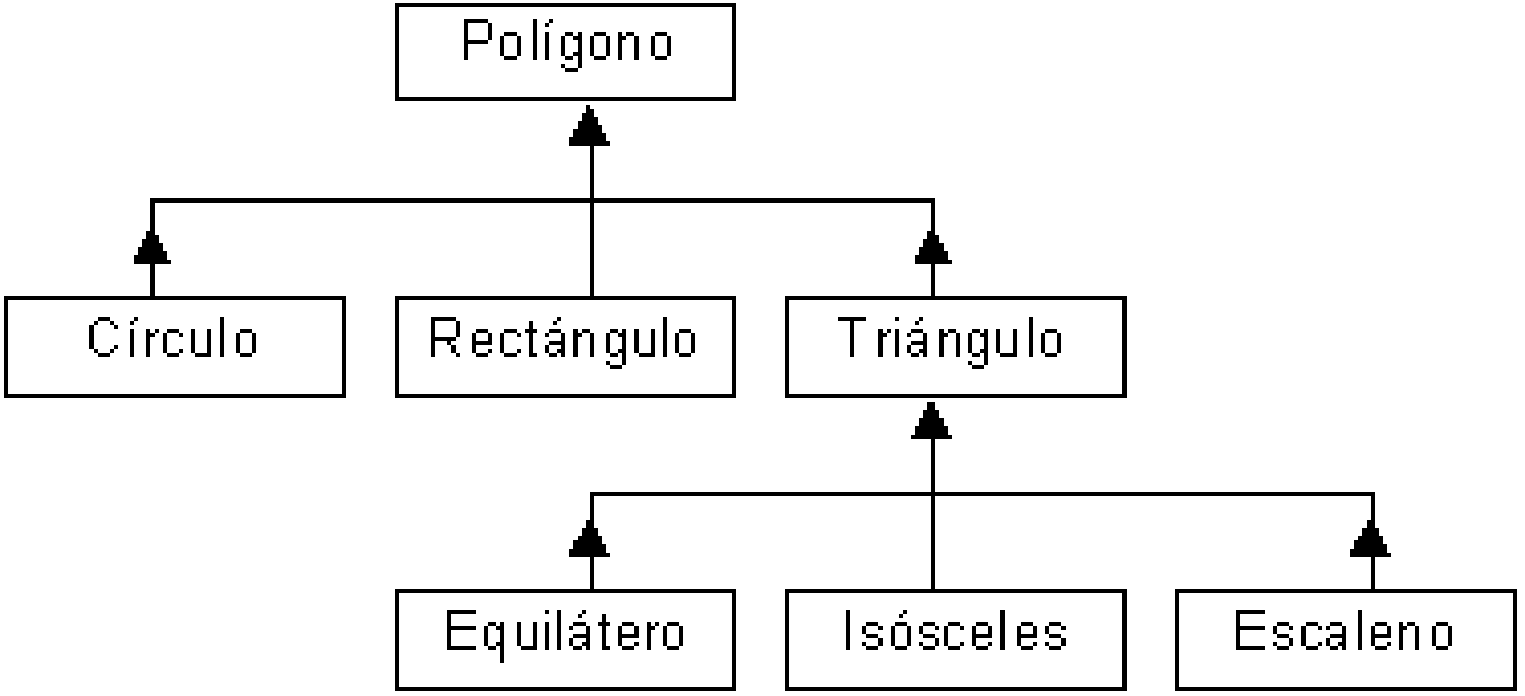
Tema 2

- Consecuencias y complejidad:
  - Es la relación más interdependiente de todas:
    - Un cambio interno en la clase base (A) afecta directamente también a las clases derivadas.
    - La clase derivada tiene que conocer la implementación de la clase base.

Tema 2 (Ejemplo)



Tema 2 (Ejemplo 2)



Tema 2 (Ejemplo java)

```java
class Persona {
    protected String nombre;
    protected String sexo;

    public String getNombre() {
        return this.nombre;
    }

    public String getSexo() {
        return this.sexo;
    }
}

class Cliente extends Persona {
    protected int numCliente;

    public int getNumCliente() {
        return this.numCliente;
    }
}

class ClientePremium extends Cliente {
    private int numFactura;

    private String getNumFactura() {
        return this.numFactura;
    }
}
```

Fijaos en el uso de **protected**

Tema 2

- Clases abstractas: no se pueden instanciar. Pueden definir métodos sin dar implementaciones.

```java
public abstract class Figura {

    // Atributos:
    private int numeroLados;
    private int area;
    private int volumen;

    // Métodos:
    abstract public void calcularArea();
    abstract public void calcularVolumen();

    public int getNumeroLados() {
        return this.numeroLados;
    }
}
```

```java
public class Esfera extends Figura {

    private float radio;

    public Esfera(int radio) {
        this.radio = radio;
        this.numeroLados = 0;
    }

    // 4*pi*r²
    public void calcularArea() {

        this.area = (4) * Math.PI * radio * radio;
    }

    // (4/3)*pi*radio³
    public void calcularVolumen() {

        this.volumen = (4 / 3) * Math.PI * radio * radio *
radio;
    }
}
```

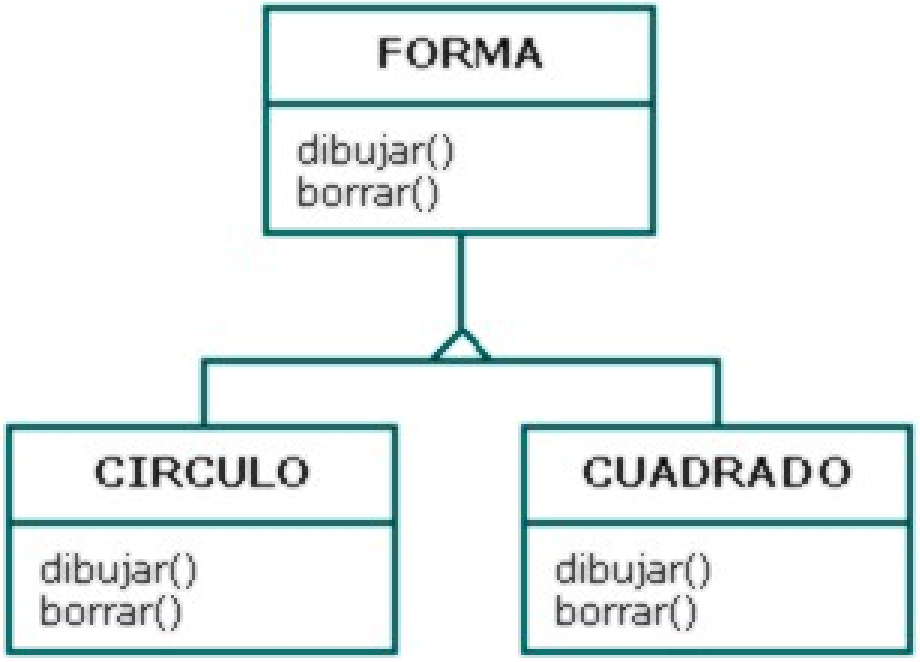
Tema 2

- Clases abstractas:
  - Las clases que hereden de la clase Abstracta deben implementar todos los métodos abstractos.
  - Se debe tener presente que las clases abstractas sí pueden heredar de otras clases.
  - La firma o parámetros de los métodos así como el tipo de dato deben respetarse, de lo contrario se está hablando de otro método totalmente diferente.
  - Si una clase abstracta está compuesta sólo por métodos abstractos y constantes entonces podemos hablar de una interface.

Tema 2

- Polimorfismo:
  - Esta característica permite definir distintos comportamientos para un método dependiendo de la clase sobre la que se realice la implementación.
  - Por ejemplo, dada una clase Vehiculo, la característica de polimorfismo habilita al programador para definir métodos en sus subclases (por ejemplo: Coche, Camión y Moto)
  - No importa qué tipo de Vehiculo sea ya que si llamamos al método getNumeroRuedas() llamará al propio de cada subclase, pero el objeto no deja de ser Vehiculo también.

Tema 2. Ejemplo



Tema 2. Ejemplo

```
abstract class Forma {
    abstract public void dibujar();
}


class Circulo extends Forma {
    public void dibujar() {
        System.out.println("Esto es un círulo");
    }
}


class Cuadrado extends Forma {
    public void dibujar() {
        System.out.println("Esto es un cuadrado");
    }
}
```

```
public class Main {
    public static void main (String [] args) {
        Forma circulo = new Circulo();
        circulo.dibujar();

        Forma cuadrado = new Cuadrado();
        cuadrado.dibujar();
    }
}
```

Tema 2

- Enlace estático: nos indica, para una variable, qué métodos está permitido invocar. Es en tiempo de compilación.
  - Ejemplo: Forma f; ← Nos indica que podemos invocar los métodos dibujar() y borrar()
- Enlace dinámico; nos indica cual implementación del método invocado se va a ejecutar. Es en tiempo de ejecución.
  - Ejemplo: Forma f = new Circulo(); ← Indica que invocaremos los métodos dibujar() y borrar() de la clase Circulo.