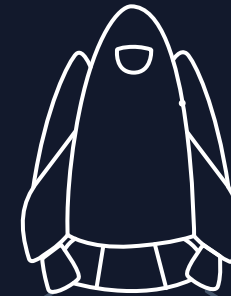


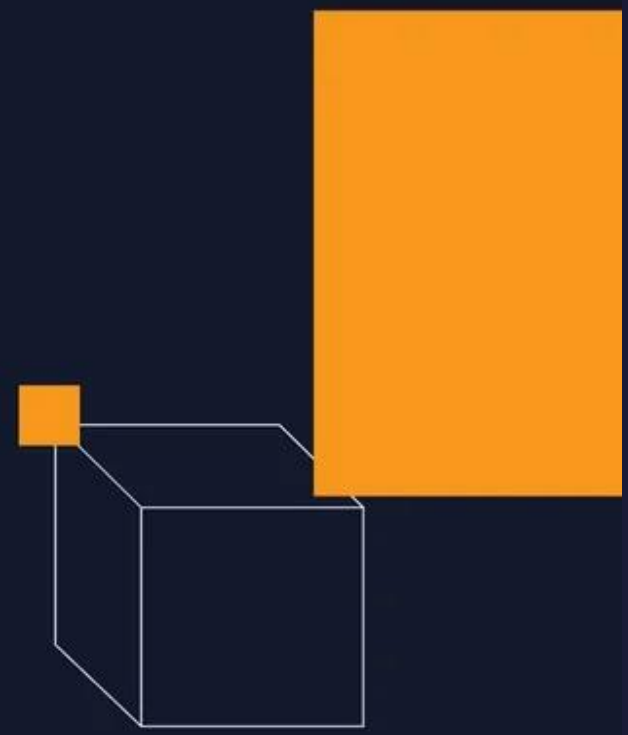
Programa académico CAMPUS



MODULO JAVA Sesión 9 **Relaciones entre clases**

Trainer Carlos H. Rueda C.



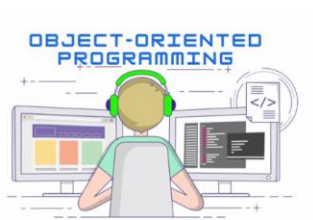




Relaciones entre clases

Las relaciones entre clases en programación orientada a objetos se refieren a cómo las clases interactúan y se asocian entre sí. Las relaciones más comunes son asociación, agregación, composición y herencia. Estas relaciones se representan con flechas en diagramas de clases para mostrar cómo las clases están conectadas.

Para complementar el concepto de las relaciones entre clases es importante conocer que es la multiplicidad pues será utilizada siempre en las relaciones entre clases.



Multiplicidad

La multiplicidad indica cuántas instancias de una clase pueden estar asociadas con una instancia de otra clase en una relación. La multiplicidad se indica en los diagramas de clases mediante números y/o un asterisco.

| Multiplicidad | Simbología |
|------------------|------------|
| Uno y solo uno | 1 |
| Cero o uno | 0 .. 1 |
| De "N" hasta "M" | N .. M |
| Varios | * |
| Cero a varios | 0 .. * |
| Uno a varios | 1 .. * |



Asociación

La asociación es una relación débil entre dos clases, donde un objeto de una clase está relacionado con uno o más objetos de otra clase, pero cada objeto existe independientemente del otro y pueden existir de manera independiente. La asociación puede ser bidireccional o unidireccional, lo que se refiere a cómo se establece la relación entre dos clases y la capacidad de navegación entre ellas.



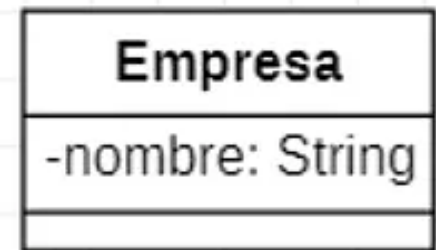
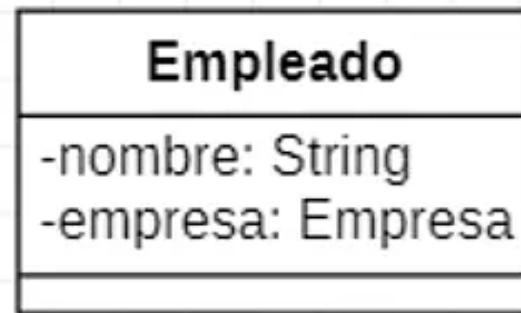
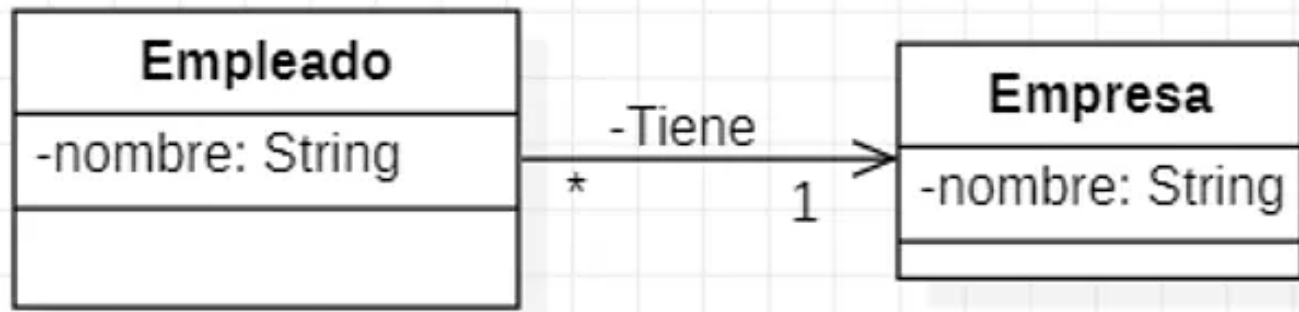
Asociación unidireccional

En una asociación unidireccional, solo una de las clases está relacionada con la otra, y la otra clase no tiene conocimiento de la relación existente. En otras palabras, una clase tiene una referencia al objeto de la otra clase, pero no viceversa.

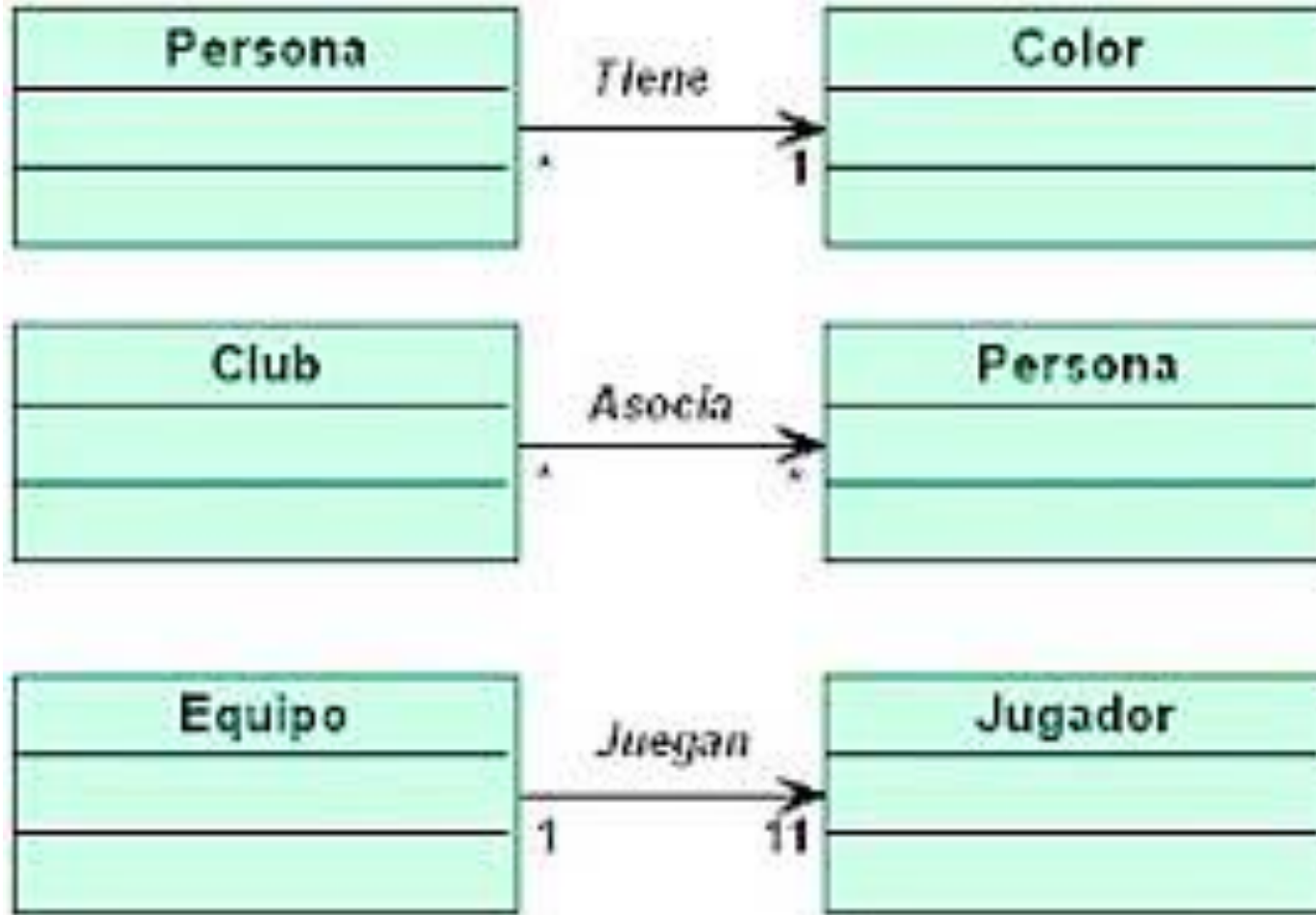


Asociación

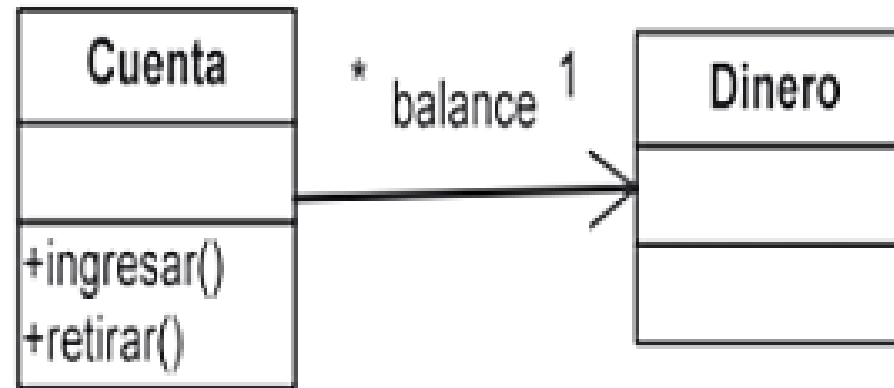
Supongamos que existen las clases Empresa y Empleado. Un empleado está asociado con una empresa, pero la empresa no necesita conocer a todos los empleados. La relación se establece desde Empleado hacia Empresa.



Asociación



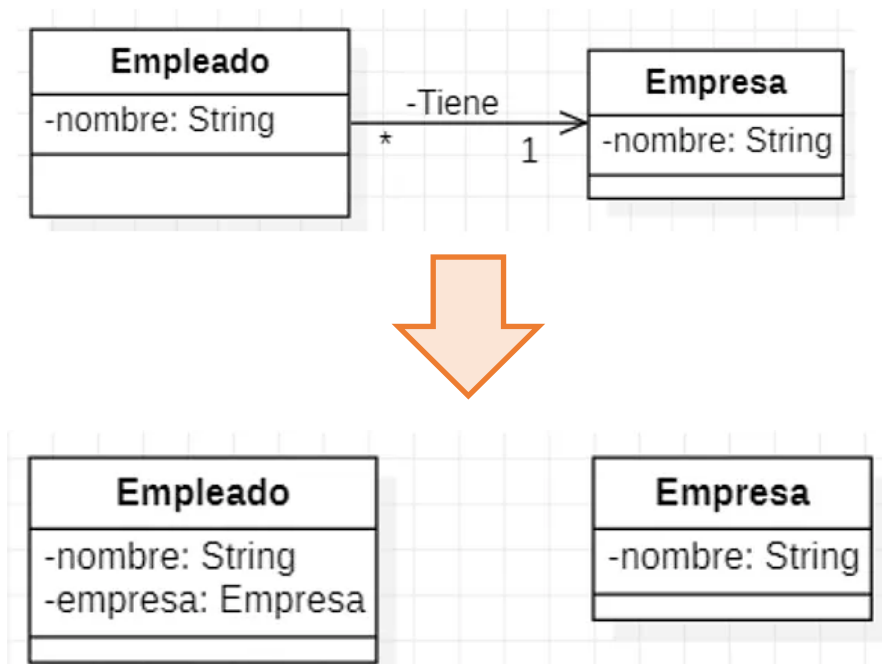
Asociación



Asociación unidireccional



Asociación

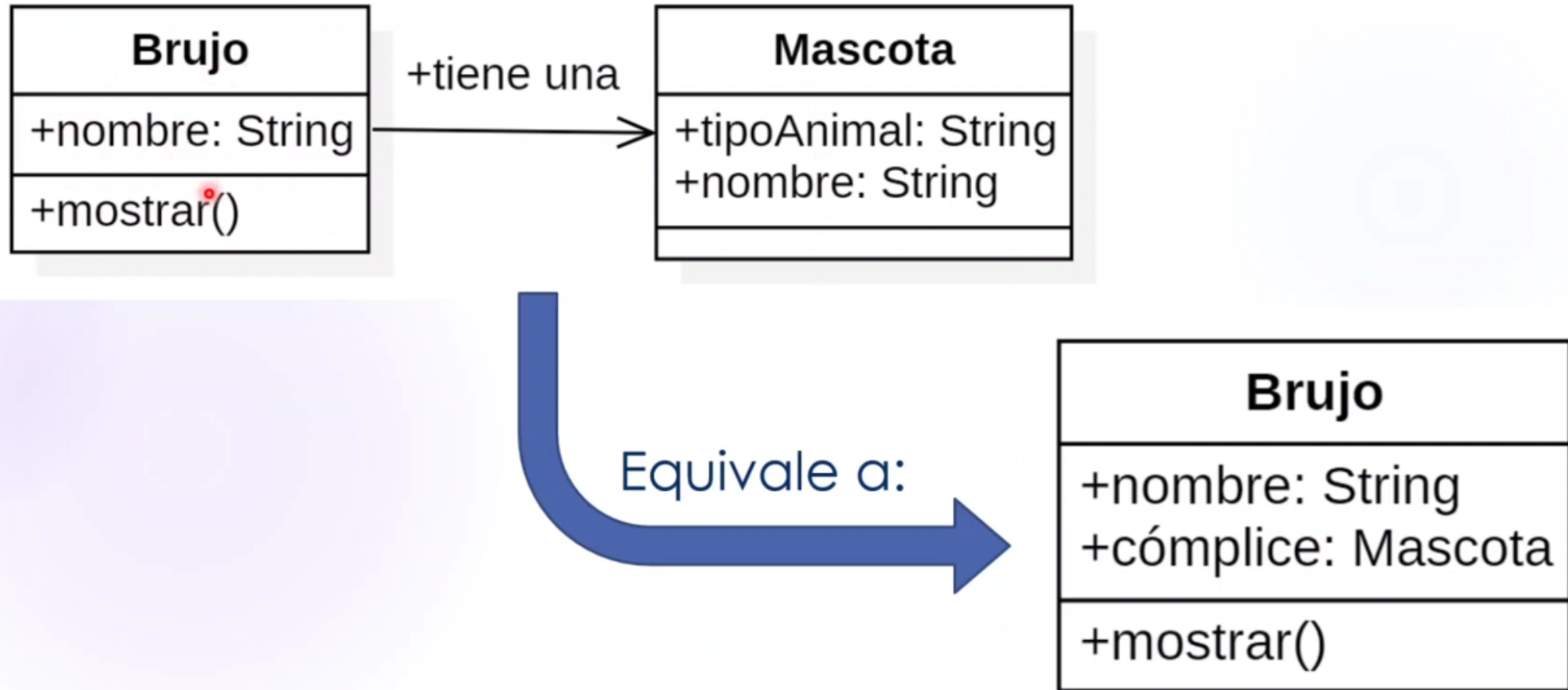


```
class Empresa {  
    private String nombre;  
    // Constructor, getters y setters  
}
```

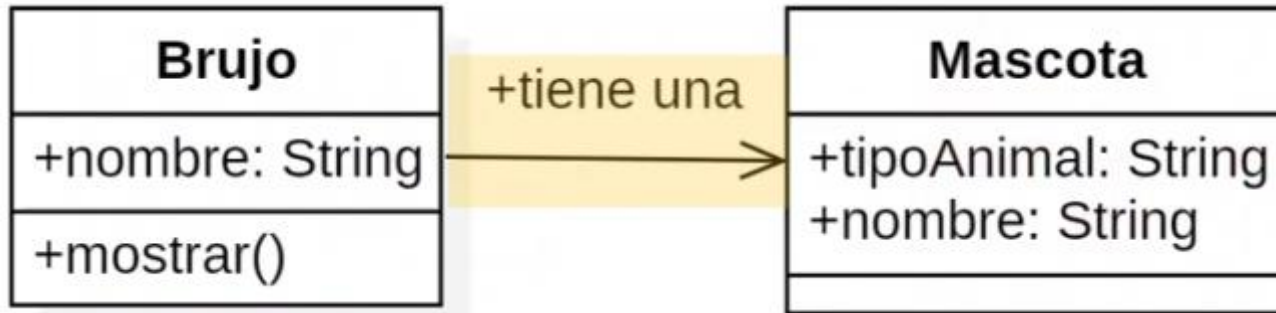
```
class Empleado {  
    private String nombre;  
    private Empresa empresa;  
  
    // Constructor, getters y setters  
}
```



Asociación



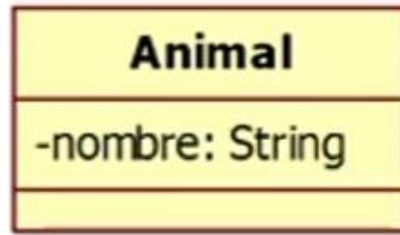
Asociación



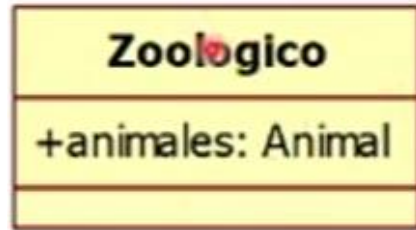
```
public class Brujo {
    public String nombre;
    public Mascota cómplice;
    public void mostrar() { ... 3 lines }
}
```



Asociación



```
public class Animal {  
    private String nombre;  
    public Animal(String nombre) {  
        this.nombre = nombre;  
    }  
}
```



```
public class Zoológico {  
    Animal [] animales;  
}
```



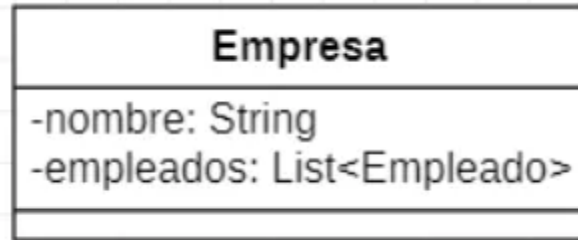
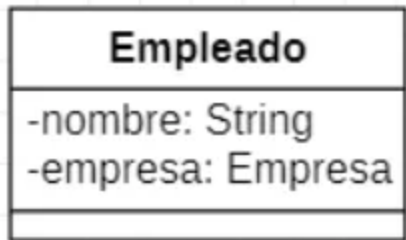
```
//declaración de un zoológico  
Zoológico chapultepec;  
//asignación de memoria (creamos el zoológico)  
chapultepec = new Zoológico();  
//creamos el arreglo de animales  
//(quedamos en clase que eran como las jaulas)  
chapultepec.animales = new Animal[3];  
//creamos los animales  
Animal a1 = new Animal("elefante");  
Animal a2 = new Animal("cocodrilo");  
Animal a3 = new Animal("tigre");  
chapultepec.animales[0] = a1;  
chapultepec.animales[1] = a2;  
chapultepec.animales[2] = a3;
```

Asociación bidireccional

En una asociación bidireccional, ambas clases están relacionadas entre sí y tienen conocimiento mutuo de la relación. Es decir, cada clase tiene una referencia al objeto de la otra clase. Siguiendo el mismo ejemplo de Empresa y Empleado, en una asociación bidireccional, la clase Empresa tendría una lista de empleados y cada empleado tendría una referencia a la empresa a la que pertenece.



Asociación bidireccional



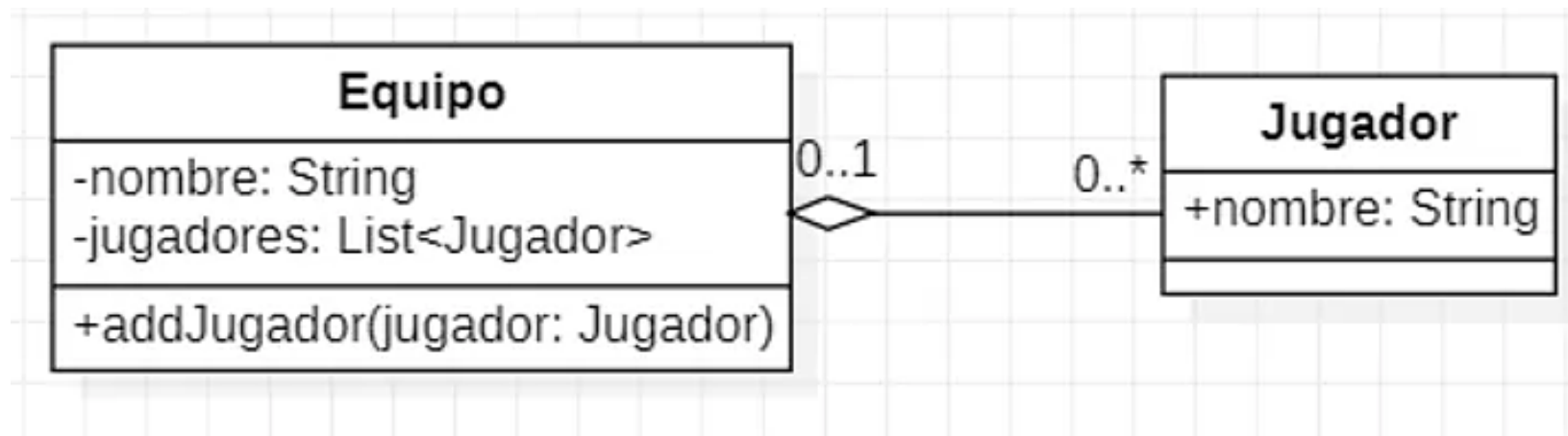
```
class Empresa {  
    private String nombre;  
    private List<Empleado> empleados;  
  
    // Constructor, getters y setters  
}  
  
class Empleado {  
    private String nombre;  
    private Empresa empresa;  
  
    // Constructor, getters y setters  
}
```



Agregación

La agregación es una relación más fuerte que la asociación. En la agregación, una clase contiene o "agrega" objetos de otra clase, pero estos objetos pueden existir independientemente de la clase contenedora. Si la clase contenedora se destruye, los objetos agregados aún pueden existir.

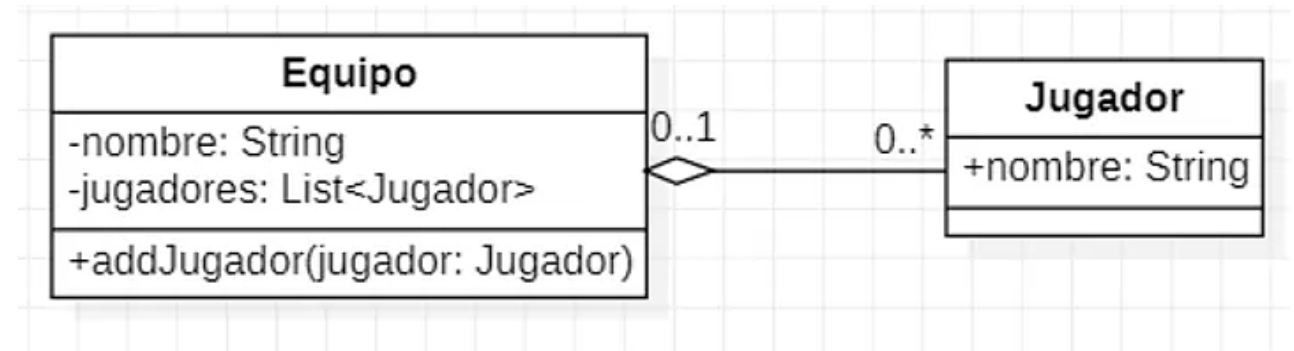
Supongamos que existen las clases Equipo y Jugador. Un equipo puede tener varios jugadores, pero los jugadores pueden existir incluso si el equipo no existe.



Agregación

```
class Equipo {  
    private String nombre;  
    private List<Jugador> jugadores;  
  
    public void addJugador(Jugador jugador){  
        jugadores.add(jugador);  
    }  
    // Constructor, getters y setters  
}
```

```
class Jugador {  
    private String nombre;  
  
    // Constructor, getters y setters  
}
```



Aquí se puede evidenciar que, los jugadores son agregados a la clase equipo a través del método `addJugador`, por lo tanto, dentro de esta lista se hará referencia a cada jugador agregado y si equipo deja de existir el jugador no lo hará, pues la creación de jugador ocurrió antes de ser agregado al equipo.



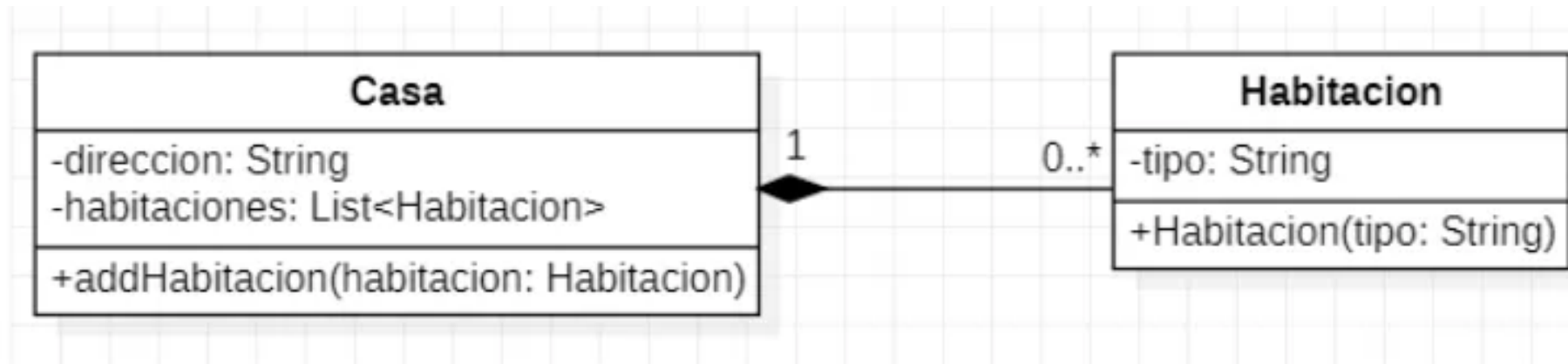
Composición

La composición es la relación más fuerte de las tres. En este caso, los objetos de una clase están completamente contenidos dentro de la clase contenedora, lo que significa que no pueden existir independientemente. Si se destruye la clase contenedora, también se destruyen los objetos contenidos.

Supongamos que existen las clases Casa y Habitación. Una casa tiene varias habitaciones, pero si la casa es demolida, todas las habitaciones también se destruyen.



Composición

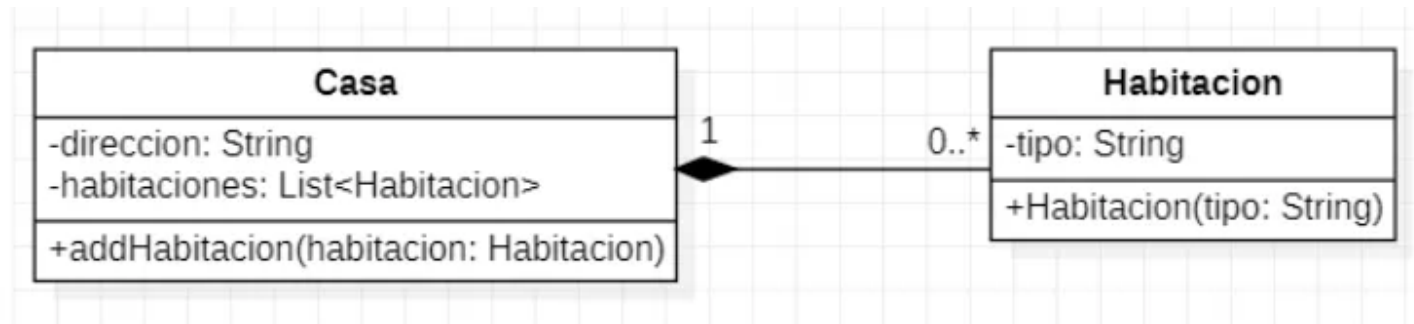


Este diagrama muestra dos clases unidas con esta particular relación que se representa con una línea con rombo negro, donde el rombo está en la clase contenedora y el otro lado en la clase contenida. Para el caso particular con la multiplicidad una casa puede tener cero o muchas habitaciones y una habitación debe estar en una casa. En código se puede ver de la siguiente forma:



Composición

```
public class Casa {  
    private String direccion;  
    private List<Habitacion> habitaciones;  
  
    public void addHabitacion(String tipo){  
        habitaciones.add(new Habitacion(tipo));  
    }  
  
    // Constructor, getters y setters  
}  
  
class Habitacion {  
    private String tipo;  
  
    public Habitacion(String tipo){  
        this.tipo = tipo;  
    }  
  
    //getters y setters  
}
```



Aquí se puede evidenciar que, las habitaciones son agregados a la clase casa a través del método `addHabitación`, pero no se agregan como objeto, sino que recibe los mismos parámetros descritos en el constructor habitación para ser creadas dentro de ese método. por lo tanto, si casa deja de existir las habitaciones lo harán pues estas últimas no fueron creadas aparte.



Composición

Panal tiene abejas; en este caso especificamos que tiene DOS abejas



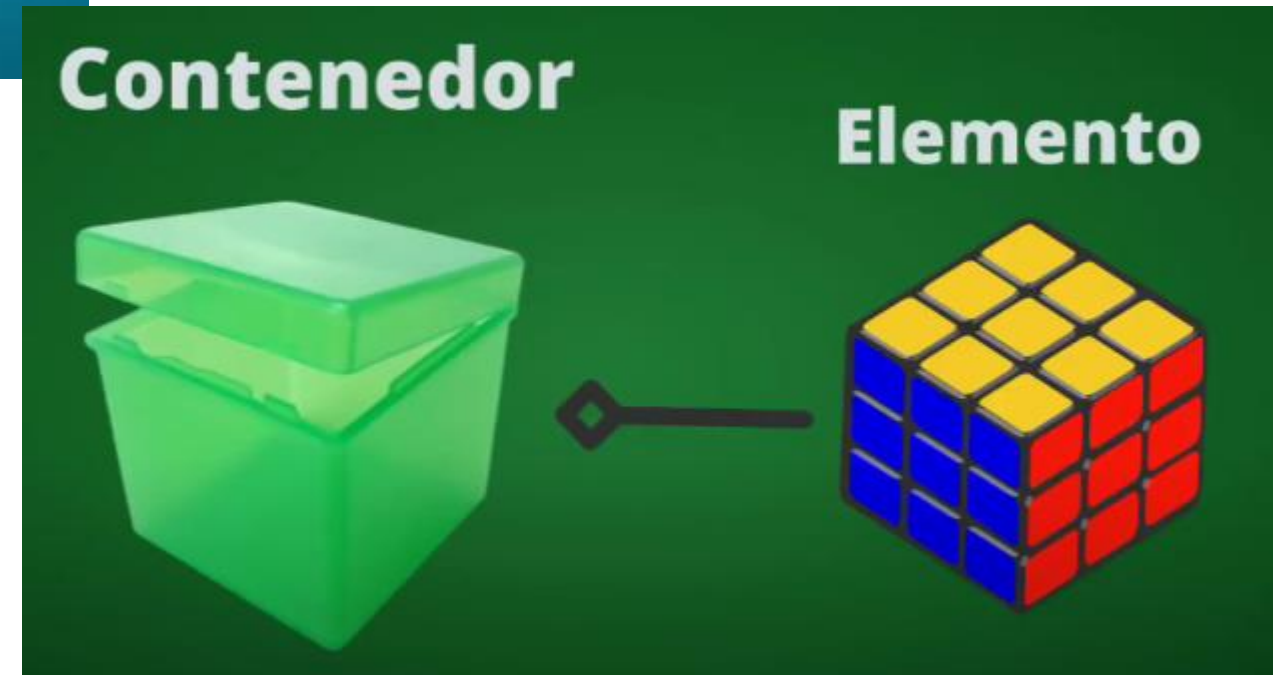
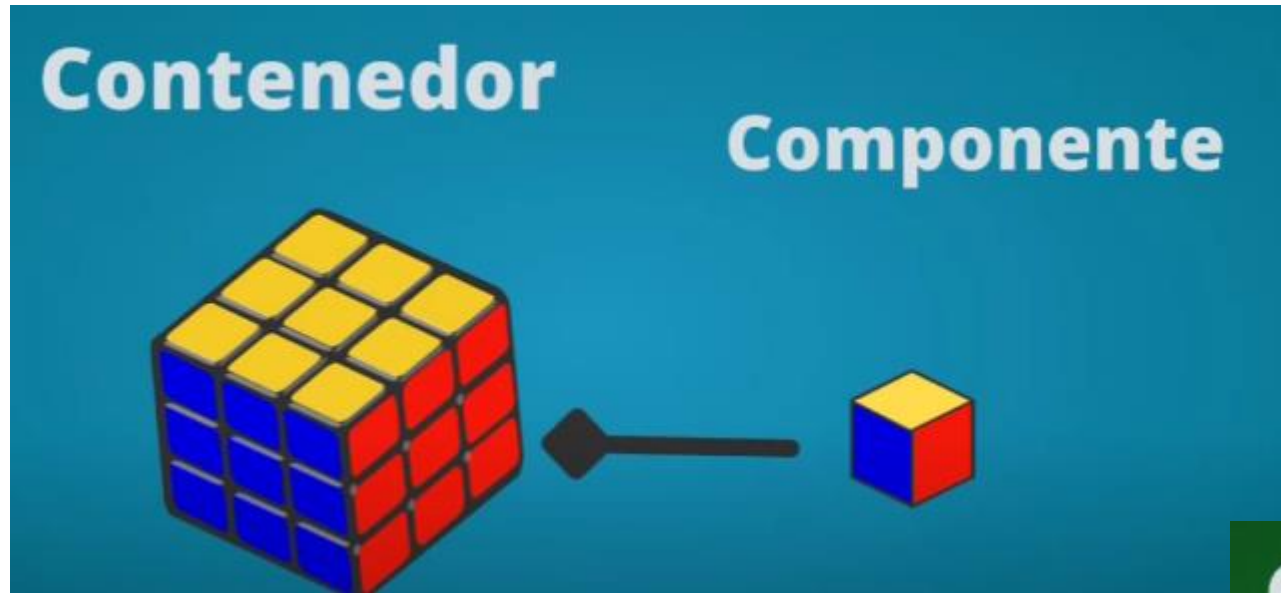
```
public class Abeja {
    private String nombre;
    public Abeja(String nombre) {
        this.nombre = nombre;
    }
}
```

```
public class Panal {
    public Abeja enjambre [];
    public Panal() {
        enjambre = new Abeja[2];
        enjambre[0] = new Abeja("Barry");
        enjambre[1] = new Abeja("Adam");
    }
}
```

```
public static void main (String args[]) {
    Panal hexagonHonexIndustries;
    hexagonHonexIndustries = new Panal();
}
```



Composición y Agregación



RELACIONES DE CLASE

▶ TIENE

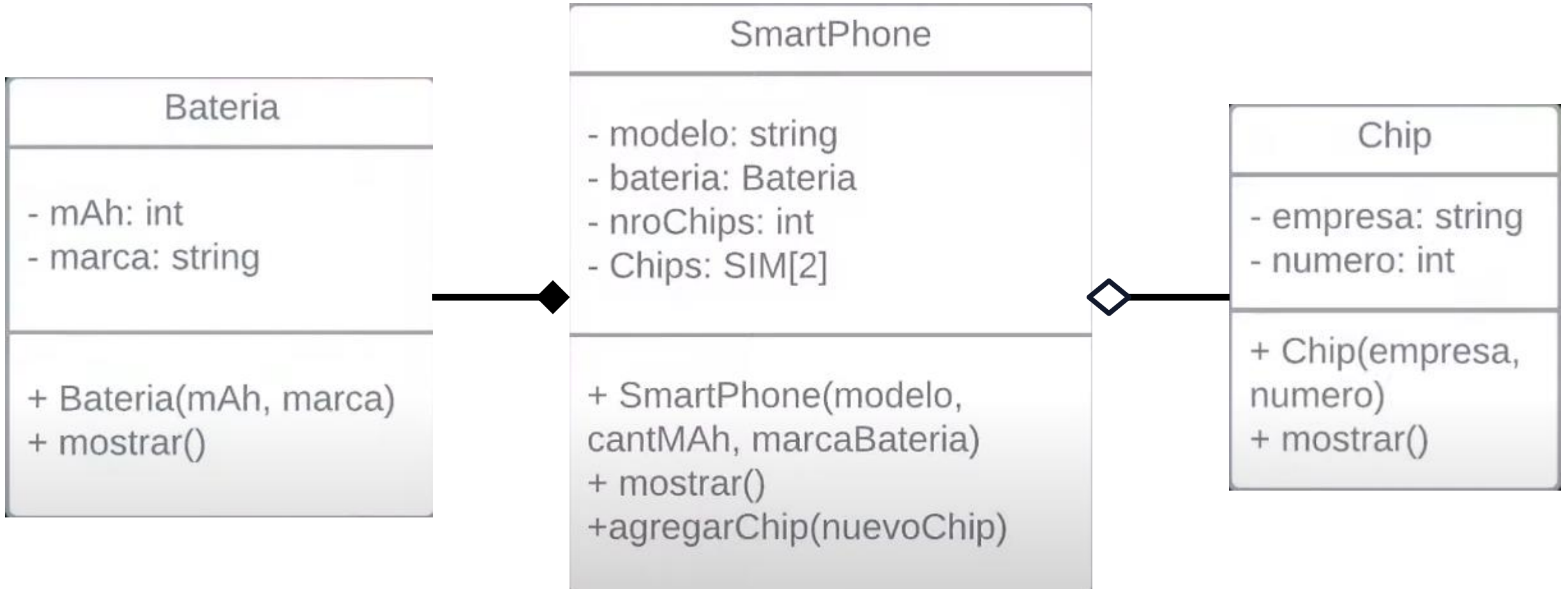
- ▶ Asociación
- ▶ Agregación (usa)
- ▶ Composición (posee)

▶ ES

- ▶ Herencia



Ejemplo



Ejemplo

```
public class Bateria {  
  
    // Atributos  
    private int mAh;  
    private String marca;  
  
    // Constructor  
    public Bateria(int mAh, String marca) {  
        this.mAh = mAh;  
        this.marca = marca;  
    }  
  
    // Metodos  
    public void mostrar() {  
        System.out.println("Cantidad mAh: " + mAh);  
        System.out.println("Marca: " + marca);  
    }  
}
```



Ejemplo

```
public class Chip {  
  
    // Atributos  
    private String empresa;  
    private int numero;  
  
    // Constructor  
    public Chip(String empresa, int numero) {  
        this.empresa = empresa;  
        this.numero = numero;  
    }  
}
```

| Chip |
|----------------------------------------|
| - empresa: string - numero: int |
| + Chip(empresa, numero) + mostrar() |



Ejemplo

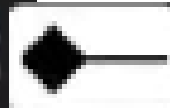
```
public class Chip {  
  
    // Atributos  
    private String empresa;  
    private int numero;  
  
    // Constructor  
    public Chip(String empresa, int numero) {  
        this.empresa = empresa;  
        this.numero = numero;  
    }  
}
```

| Chip |
|----------------------------------------|
| - empresa: string - numero: int |
| + Chip(empresa, numero) + mostrar() |



Ejemplo

```
public class SmartPhone {  
  
    // Atributos  
    private String modelo;  
    private Bateria bateria;  
    private int nroChips;  
    private Chip[] chips;  
  
}
```





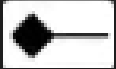

SmartPhone

- modelo: string
- bateria: Bateria
- nroChips: int
- Chips: SIM[2]

- + SmartPhone(modelo, cantMAh, marcaBateria)
- + mostrar()
- + agregarChip(nuevoChip)



Ejemplo

```
public class SmartPhone {  
  
    // Atributos  
    private String modelo;  
    private Bateria bateria;   
    private int nroChips;  
    private Chip[] chips;   
  
    // Constructor  
    public SmartPhone(String modelo, int cantMAh, String marcaBateria) {  
        this.modelo = modelo;  
        this.bateria = new Bateria(cantMAh, marcaBateria);   
        this.nroChips = 0;  
        this.chips = new Chip[2];   
    }  
}
```



Ejemplo

```
// Metodos
public void mostrar() {
    System.out.println("Modelo: " + modelo);
    bateria.mostrar();
    System.out.println("Nro Chips: " + nroChips);
    for (int i = 0; i < nroChips; i++) {
        chips[i].mostrar();
    }
}

public void agregarChip(Chip nuevoChip) {
    if (nroChips < 2) {
        chips[nroChips] = nuevoChip;
        nroChips++;
    }
}
```



Ejemplo

```
public static void main(String[] args) {  
  
    SmartPhone cel = new SmartPhone("A10PRO", 3400, "Samsung");  
  
    Chip entel = new Chip("Entel", 79033881);  
    Chip tigo = new Chip("Tigo", 69920081);  
  
    cel.agregarChip(entel);  
    cel.agregarChip(tigo);  
  
    cel.mostrar();  
  
}
```



Ejercicio

En una universidad, se requiere gestionar información básica de las personas que forman parte de ella. Estas personas pueden ser **estudiantes**, **profesores** o **administrativos**. Toda persona tiene un nombre, un apellido y una fecha de nacimiento, que consta de un día, un mes y un año.

Además, se llevan a cabo **actividades** dentro de la universidad, y cada actividad tiene una descripción y un participante asociado, que puede ser cualquier persona de la universidad.

El sistema debe permitir crear personas con sus respectivos datos y registrar actividades asociadas a ellas. Finalmente, debe ser posible mostrar la información de las personas y las actividades en las que participan.

Realiza el diagrama UML que modela el ejercicio. Luego, desarrolla las clases del modelo e implementa un código de prueba de dichas clases. El programa debe permitir:

- ✓ Crear una universidad y los participantes.
- ✓ Asignar estudiantes y profesores.
- ✓ Representar actividades en las que participen las personas.
- ✓ Mostrar la información de la universidad y las actividades de manera detallada.

Ejercicios

Calcula la letra de un DNI, pediremos el DNI por teclado y nos devolverá el DNI completo. Para calcular la letra, cogeremos el resto de dividir nuestro DNI entre 23, el resultado debe estar entre 0 y 22. Haz un método donde según el resultado de la anterior formula busque en un array de caracteres la posición que corresponda a la letra. Por ejemplo, si introduzco 70588387, el resultado será de 7 que corresponde a 'F'.

El listado de la tabla está en la siguiente tabla:

Las personas del ejercicio anterior poseen un DNI.

- ✓ Modifica el diagrama UML anterior para que se agregue este nuevo requerimiento.
- ✓ Modifica el programa anterior para que permita calcular, agregar y mostrar el DNI a las personas.

| Posicion | Letra | Posicion | Letra | Posicion | Letra |
|----------|-------|----------|-------|----------|-------|
| 0 | T | 11 | B | 21 | K |
| 1 | R | 12 | N | 22 | E |
| 2 | W | 13 | J | | |
| 3 | A | 14 | Z | | |
| 4 | G | 15 | S | | |
| 5 | M | 16 | Q | | |
| 6 | Y | 17 | V | | |
| 7 | F | 18 | H | | |
| 8 | P | 19 | L | | |
| 9 | D | 20 | C | | |
| 10 | X | | | | |



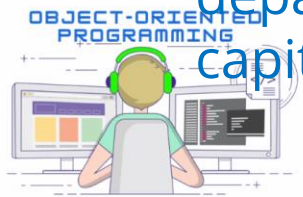
Ejercicios

Cree un programa en el cual almacenará el listado de los departamentos de Colombia, posteriormente realizar las siguientes acciones:

1. Mostrar en pantalla el primer y último departamento almacenado en el vector.
2. Conocer en qué posición se encuentra determinado departamento en el vector (Con su nombre).
3. Se debe poder eliminar una ciudad del vector, sin perder el orden del mismo.
4. Mostrar todos los departamentos almacenados en el vector.

La diferentes opciones deben poder accederse a través de un menú. Valide las opciones e información ingresada por el usuario.

- ✓ Diseñe correctamente los requerimientos usando UML
- ✓ Cree una entidad Pais donde se pueda agregar los departamentos
- ✓ Cree una aplicación que permita agregar ciudades, departamentos, listar todos los departamentos, todas las ciudades, todas las ciudades de un departamento, la ciudad capital de un departamento, los departamentos con sus capitales y todas las capitales.



Ejercicios

Se quiere simular un juego en el que participan N jugadores y otra persona que hace de árbitro. Cada jugador elige 4 números en el rango $[1, 10]$, pudiendo estar repetidos. A continuación, el árbitro, sin conocer los números que ha elegido cada jugador, selecciona 2 números A y B .

El programa debe ser capaz de calcular cuántos números de los seleccionados por cada jugador están comprendidos entre los valores A y B . Ganará el jugador que más números tenga en dicho intervalo.

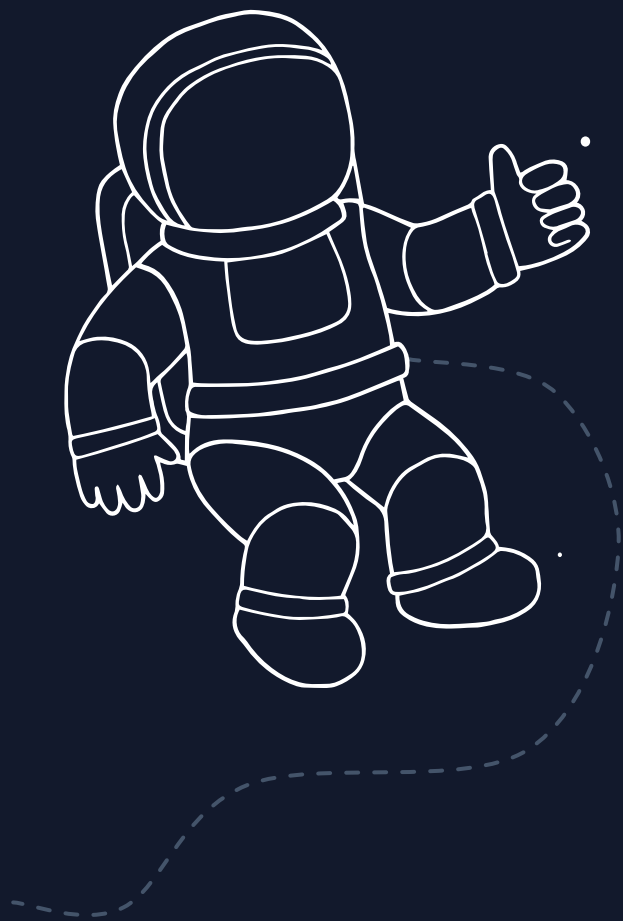
Se pide implementar un programa modular que simule el juego para 3 jugadores, teniendo en cuenta que:

- Tanto los 4 datos de cada jugador, como los valores para A y B se introducirán por teclado. En todos los casos, el programa detectará la entrada de números erróneos, solicitando nuevamente el dato hasta que sea válido.
- Se deben mostrar por pantalla no solo los aciertos de cada jugador sino los datos que ha introducido cada jugador y los que ha seleccionado el árbitro. Por último, hay que imprimir la media aritmética de los aciertos de todos los jugadores

- ✓ Realice el diseño UML de las clases del juego y de las relaciones entre ellas. No olvide colocar la cardinalidad.







Programa acadêmico CAMPUS

Módulo JAVA

