

# Introducción a la Programación Orientada a Objetos

Unidad: Programación Orientada a Objetos (POO) - Sesión 2

Introducción a la Programación

# Programación Orientada a Objetos — Sesión 2

## Arreglos de objetos y clase Agenda

- ▶ En la sesión anterior:
  - ▶ Definimos clases y objetos (Contacto, Butaca).
  - ▶ Vimos atributos, métodos, encapsulamiento y constructores.
- ▶ Hoy daremos un paso más:
  - ▶ Trabajar con **arreglos de objetos** (Contacto[]).
  - ▶ Diseñar una clase que **gestione** varios objetos: AgendaContactos.
  - ▶ Ver la idea de **composición**: una agenda *tiene* muchos contactos.
- ▶ Conexión con los laboratorios:
  - ▶ Lab 1: una butaca como objeto individual.
  - ▶ Lab 2: una función de cine con **muchas butacas** usando arreglos de objetos.

# Recordatorio: clase Contacto

Ejemplo trabajado en la sesión anterior

- ▶ En clase vimos una clase Contacto con:
  - ▶ Atributos privados: nombre, telefono.
  - ▶ Constructor para inicializar los datos.
  - ▶ Método imprimirResumen().

```
class Contacto {  
    private String nombre;  
    private String telefono;  
    public Contacto(String nombre, String telefono) {  
        this.nombre = nombre;  
        this.telefono = telefono;  
    }  
    public void imprimirResumen() {  
        System.out.println(nombre + " - " + telefono);  
    }  
}
```

- ▶ Hoy vamos a pasar de **un contacto** a **muchos contactos**, y a diseñar una clase que los gestione como una agenda.

# De objetos sueltos a arreglos de objetos

Cuando un solo Contacto ya no alcanza

- ▶ Con un solo contacto, el uso es sencillo:

```
Contacto c1 = new Contacto("Ana Lopez", "7777777");  
c1.imprimirResumen();
```

- ▶ Pero una agenda real tiene muchos contactos:

```
Contacto c1 = new Contacto("Ana Lopez", "7777777");  
Contacto c2 = new Contacto("Carlos Perez", "8888888");  
Contacto c3 = new Contacto("Luis Gomez", "9999999");  
// ...
```

- ▶ Problema:

- ▶ El código se vuelve repetitivo y difícil de mantener.
- ▶ No hay una forma sencilla de **recorrer** todos los contactos.

- ▶ Solución:

- ▶ Usar un **arreglo de objetos** Contacto[] para guardar varios contactos y poder recorrerlos con un bucle.

# Primer ejemplo con Contacto[]

## Arreglo de objetos con tamaño fijo

- Un arreglo de contactos se declara así:

```
Contacto[] contactos = new Contacto[5]; // hasta 5 contactos
```

- Al principio, todas las posiciones están en null:

```
// Crear algunos contactos y guardarlos en el arreglo
contactos[0] = new Contacto("Ana Lopez", "7777777");
contactos[1] = new Contacto("Carlos Perez", "8888888");
// Recorrer las posiciones ocupadas
for (int i = 0; i < contactos.length; i++) {
    if (contactos[i] != null) {
        contactos[i].imprimirResumen();
    }
}
```

- Idea clave:

- **Arreglo de objetos** = estructura que guarda varias instancias de una misma clase.

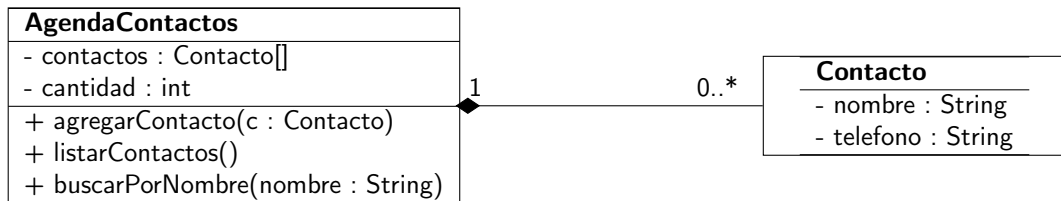
# Limitaciones de usar solo Contacto []

Necesitamos una clase que gestione la agenda

- ▶ Si solo tenemos un Contacto [] en main:
  - ▶ Todo el manejo de la agenda queda mezclado en el main.
  - ▶ Se repite código para:
    - ▶ Agregar contactos.
    - ▶ Buscar un contacto por nombre.
    - ▶ Listar todos los contactos.
- ▶ Problemas:
  - ▶ El main se vuelve muy largo y difícil de leer.
  - ▶ No hay un lugar claro donde estén las **reglas de la agenda**.
- ▶ Mejor diseño:
  - ▶ Crear una clase AgendaContactos que:
    - ▶ Tenga internamente un Contacto [].
    - ▶ Ofrezca métodos para trabajar con la agenda.
  - ▶ **Composición**: una agenda *tiene* muchos contactos.

# Composición en UML: AgendaContactos y Contacto

Una agenda *tiene* muchos contactos



- ▶ AgendaContactos **compone** varios Contacto: contiene el arreglo y controla su ciclo de vida.
- ▶ Esto es un ejemplo de **composición** en UML: si la agenda desaparece, sus contactos internos también.

# Esqueleto de la clase AgendaContactos

Una clase que gestiona varios contactos

- ▶ Vamos a diseñar una clase que:
  - ▶ Encapsule el arreglo Contacto[].
  - ▶ Lleve la cuenta de cuántos contactos hay guardados.

```
class AgendaContactos {  
    private Contacto[] contactos;  
    private int cantidad; // contactos actualmente registrados  
    public AgendaContactos(int capacidadMaxima) {  
        contactos = new Contacto[capacidadMaxima];  
        cantidad = 0;  
    }  
    // Aqui iran metodos como:  
    // agregarContacto(...)  
    // listarContactos()  
    // buscarPorNombre(...)  
}
```

- ▶ Desde main, solo usaremos la clase AgendaContactos para interactuar con los contactos.



# Clase AgendaContactos: atributos y constructor

## La agenda como contenedor de contactos

- ▶ AgendaContactos se encarga de:
  - ▶ Guardar varios objetos Contacto.
  - ▶ Recordar cuántos contactos hay registrados.
- ▶ Para eso, necesita:
  - ▶ Un arreglo Contacto[] con capacidad fija.
  - ▶ Un contador cantidad.

```
class AgendaContactos {  
    private Contacto[] contactos;  
    private int cantidad; // contactos actualmente registrados  
    public AgendaContactos(int capacidadMaxima) {  
        contactos = new Contacto[capacidadMaxima];  
        cantidad = 0;  
    }  
    // Aqui iran metodos como:  
    // agregarContacto(...)  
    // listarContactos()  
    // buscarPorNombre(...)  
}
```

# Método agregarContacto

Agregar un contacto a la agenda

- ▶ Queremos un método que:
  - ▶ Verifique si hay espacio en el arreglo.
  - ▶ Guarde el contacto en la siguiente posición libre.
  - ▶ Actualice el contador cantidad.
  - ▶ Indique con true/false si se pudo agregar.

```
class AgendaContactos {  
    // atributos y constructor ...  
    public boolean agregarContacto(Contacto c) {  
        if (cantidad < contactos.length) {  
            contactos[cantidad] = c;  
            cantidad++;  
            return true;  
        } else {  
            // agenda llena  
            return false;  
        }  
    }  
}
```

# Método listarContactos()

Recorrer y mostrar todos los contactos

- ▶ La clase AgendaContactos debe saber:
  - ▶ Cuántos contactos hay (cantidad).
  - ▶ Cómo mostrarlos en pantalla.

```
class AgendaContactos {  
    // ...  
    public void listarContactos() {  
        System.out.println("Contactos en la agenda:");  
        for (int i = 0; i < cantidad; i++) {  
            contactos[i].imprimirResumen();  
        }  
    }  
}
```

- ▶ Notar que:
  - ▶ Solo recorreremos hasta cantidad.
  - ▶ No necesitamos verificar null si mantenemos bien el contador.

## Método buscarPorNombre()

Devolver el primer contacto que coincide

- ▶ También queremos buscar un contacto por su nombre.
- ▶ Una opción: devolver el **objeto** Contacto o null si no se encuentra.

```
class AgendaContactos {  
    // ...  
    public Contacto buscarPorNombre(String nombreBuscado) {  
        for (int i = 0; i < cantidad; i++) {  
            // Aqui usamos un getter (no mostrado)  
            // o accedemos a un campo publico si fuera el caso  
            // Supongamos que existe getNombre():  
            // if (contactos[i].getNombre().equals(nombreBuscado)) { ... }  
            // Ejemplo simplificado si nombre fuera publico:  
            // if (contactos[i].nombre.equals(nombreBuscado)) { ... }  
        }  
        return null; // no se encontro  
    }  
}
```

- ▶ La idea clave es:

# Ejemplo de uso de AgendaContactos en main

Cerrando el ciclo: de objetos individuales a una agenda

```
class Principal {  
    public static void main(String[] args) {  
        AgendaContactos agenda = new AgendaContactos(5);  
        Contacto c1 = new Contacto("Ana Lopez", "77777777");  
        Contacto c2 = new Contacto("Carlos Perez", "88888888");  
        agenda.agregarContacto(c1);  
        agenda.agregarContacto(c2);  
        agenda.listarContactos();  
        // Ejemplo de busqueda (cuando exista getNombre)  
        // Contacto encontrado = agenda.buscarPorNombre("Ana Lopez");  
        // if (encontrado != null) {  
        //     System.out.println("Contacto encontrado:");  
        //     encontrado.imprimirResumen();  
        // }  
    }  
}
```

# Resumen de la sesión 2

## De objetos individuales a colecciones de objetos

- ▶ Partimos del ejemplo de clase:
  - ▶ **Contacto** como objeto con sus datos encapsulados.
  - ▶ Recordamos constructores y métodos de instancia.
- ▶ Nueva idea central:
  - ▶ Uso de **arreglos de objetos** (`Contacto []`).
  - ▶ Una clase que **gestiona** varios objetos: `AgendaContactos`.
  - ▶ Relación de **composición**: una agenda *tiene* muchos contactos.
- ▶ Buenas prácticas vistas:
  - ▶ Encapsular el arreglo dentro de la agenda (`private Contacto []`).
  - ▶ Llevar un **contador** (cantidad) en lugar de usar `null`.
  - ▶ Delegar operaciones en métodos: `agregarContacto`, `listarContactos`, `buscarPorNombre`.
- ▶ Conexión con los laboratorios:
  - ▶ **Lab 1**: diseño de una Butaca individual (POO básica).
  - ▶ **Lab 2**: una `FuncionCine` que **contiene muchas butacas**, aplicando arreglos de objetos y una clase gestora, igual que `AgendaContactos`.