

# Sesión 07: Invocación de métodos

## Programación 2

### 2. Colecciones acotadas de objetos

---

Ángel Herranz

2019-2020

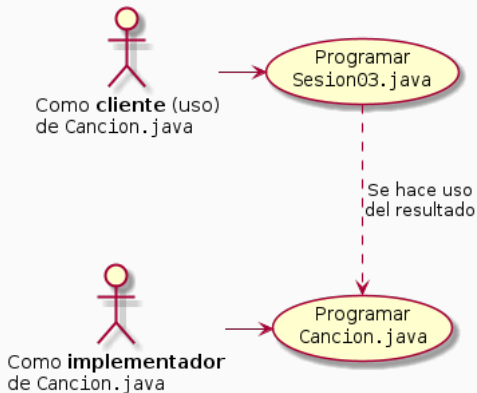
Universidad Politécnica de Madrid

# En capítulos anteriores...

- ...
  - **Objetos**, **referencias** y variables (y **primitivos**)
  - **Clases**: plantilla para crear objetos
- Encapsular**  
datos y comportamiento
- Terminología y **ocultación**
  - Modelización: racionales, puntos, naipes, ...



# Dos roles i





# Dos roles ii

## Cuando lo usas

- No quieres saber cómo está hecho
- Sólo quieres saber qué hace
- No quieres repetir trabajo

## Cuando implementas

- Estás al servicio de quienes lo usan
- Que no necesiten mirar el cómo
- Explicar claramente el qué hace



# Ocultación

Para evitar que una modificación en la representación interna de una clase afecta a quien la usa, es muy importante ocultar dicha representación



# Ocultación

Para evitar que una modificación en la representación interna de una clase afecta a quien la usa, es muy importante ocultar dicha representación

```
class Punto2D {  
    double x;  
    double y;  
    ...  
}  
  
public static void  
    main(String[] args) {  
    Punto2D p =  
        new Punto2D();  
    p.x = 1;  
    p.y = -1;  
}
```



# Ocultación

Para evitar que una modificación en la representación interna de una clase afecta a quien la usa, es muy importante ocultar dicha representación

```
class Punto2D {  
    private double x;  
    private double y;  
    ...  
}
```

```
public static void  
    main(String[] args) {  
        Punto2D p =  
            new Punto2D();  
        p.x = 1; 👍 No compila  
        p.y = -1; 👍 No compila  
    }
```

# En el capítulo de hoy...

- Tema 2. Colecciones acotadas de objetos.
- Pero antes...



¿Cómo funciona **a.f(b.g())**?

---



# Análisis sintáctico

*Colorless green ideas sleep furiously*

Noam Chomsky



# Análisis sintáctico

*Colorless green ideas sleep furiously*

Noam Chomsky



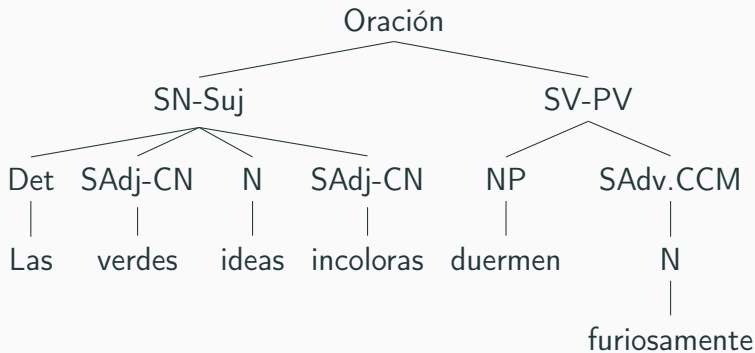
# Análisis sintáctico

Las verdes ideas incoloras duermen furiosamente

Noam Chomsky



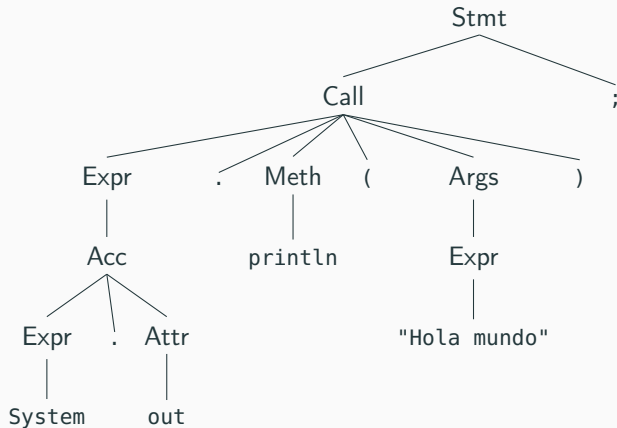
# Análisis sintáctico



# Lo mismo en Java

```
System.out.println("Hola mundo");
```

# Lo mismo en Java



`System.out.println("Hola mundo");`

# ¿Algo más complejo?

```
System.out.println(new Punto2D(0,0).distancia(p));
```

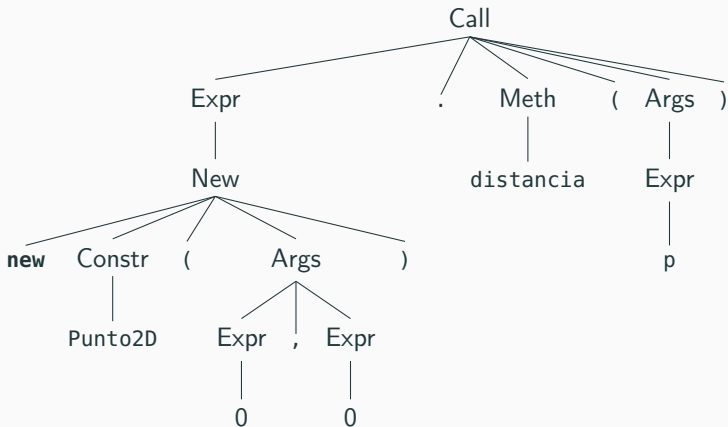


# ¿Algo más complejo?

```
new Punto2D(0,0).distancia(p)
```

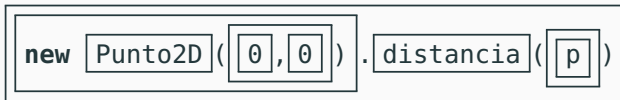
# ¿Algo más complejo?

`new Punto2D(0,0).distancia(p)`



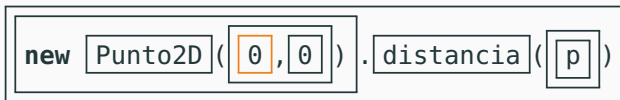
# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



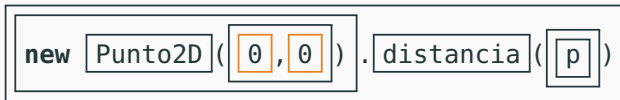
# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



# ¿Algo más complejo?

`new Punto2D(0,0).distancia(p)`



# ¿Algo más complejo?

`new Punto2D(0,0).distancia(p)`





# ¿Algo más complejo?

`new Punto2D(0,0).distancia(p)`



# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



# ¿Algo más complejo?

**new** Punto2D(0,0).distancia(p)



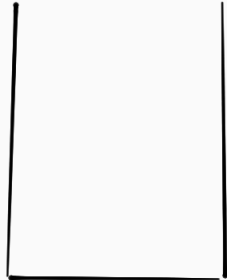
# Pilas, pilas y pilas



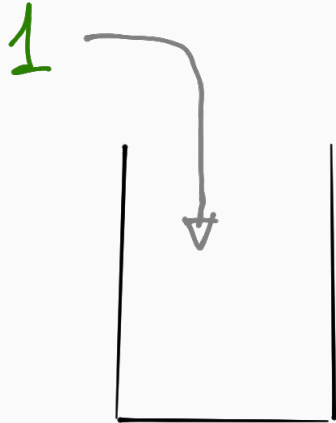
# Pilas, pilas y pilas



# Pilas, pilas y pilas (*Stack*)

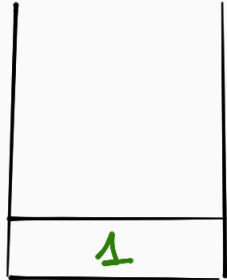


# Pilas, pilas y pilas (*Stack*)

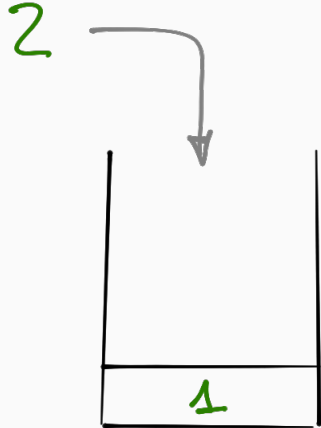




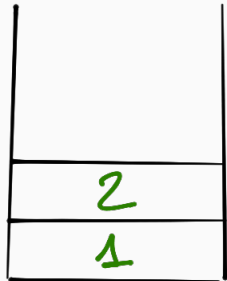
# Pilas, pilas y pilas (*Stack*)



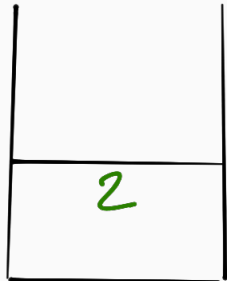
# Pilas, pilas y pilas (*Stack*)



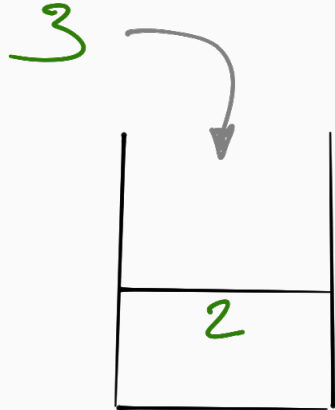
# Pilas, pilas y pilas (*Stack*)



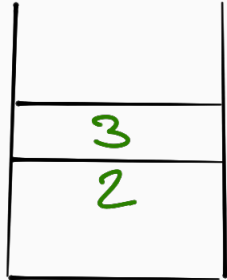
# Pilas, pilas y pilas (*Stack*)



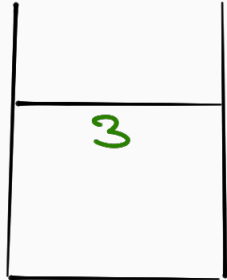
# Pilas, pilas y pilas (*Stack*)



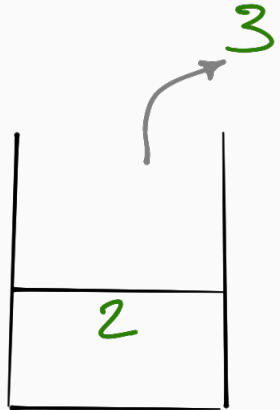
# Pilas, pilas y pilas (*Stack*)



# Pilas, pilas y pilas (*Stack*)

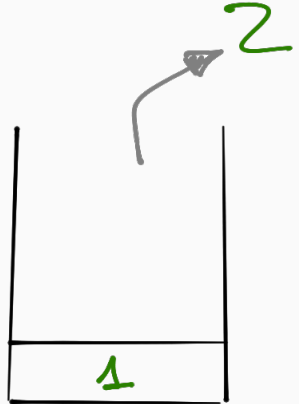


# Pilas, pilas y pilas (*Stack*)

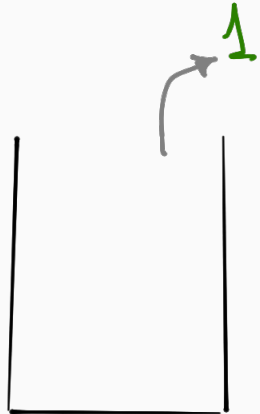




# Pilas, pilas y pilas (*Stack*)



# Pilas, pilas y pilas (*Stack*)



# ¿Qué ocurre al invocar un método?

```
public static void  
    main(String[] args) {  
  
    Punto o =  
        new Punto2D();  
  
    Punto a =  
        new Punto2D(1,1);  
  
    System.out.println(  
        a.distancia(o)  
    );  
}
```

Q: ¿Cuál es el primer método que se invoca?

# ¿Qué ocurre al invocar un método?

```
public static void  
    main(String[] args) {  
  
    Punto o =  
        new Punto2D();  
  
    Punto a =  
        new Punto2D(1,1);  
  
    System.out.println(  
        a.distancia(o)  
    );  
}
```

Herranz

Q: ¿Cuál es el primer método que se invoca?

A: **jmain!**

Q: ¿Y después? (al margen de **new**)

# ¿Qué ocurre al invocar un método?

```
public static void  
    main(String[] args) {  
  
    Punto o =  
        new Punto2D();  
  
    Punto a =  
        new Punto2D(1,1);  
  
    System.out.println(  
        a.distancia(o)  
    );  
}
```

Herranz

Q: ¿Cuál es el primer método que se invoca?

A: **jmain!**

Q: ¿Y después? (al margen de **new**)

A: **distancia**

Q: ¿Y después?

# ¿Qué ocurre al invocar un método?

```
public static void  
    main(String[] args) {  
  
    Punto o =  
        new Punto2D();  
  
    Punto a =  
        new Punto2D(1,1);  
  
    System.out.println(  
        a.distancia(o)  
    );  
}
```

Herranz

Q: ¿Cuál es el primer método que se invoca?

A: **jmain!**

Q: ¿Y después? (al margen de **new**)

A: **distancia**

Q: ¿Y después?

A: **println**

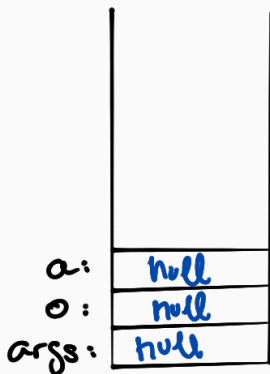
# Pila (*Stack*) de ejecución

Empieza la ejecución de main



# Pila (*Stack*) de ejecución

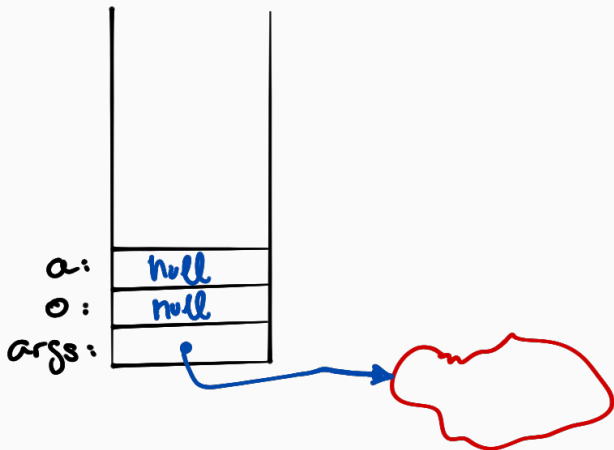
Espacio para argumentos y variables locales





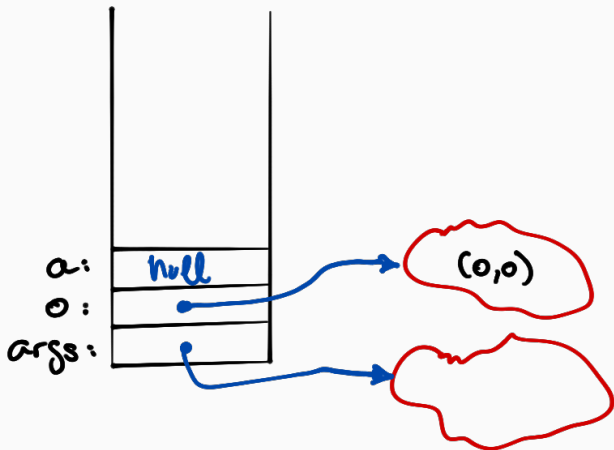
# Pila (*Stack*) de ejecución

Se cargan los argumentos del main



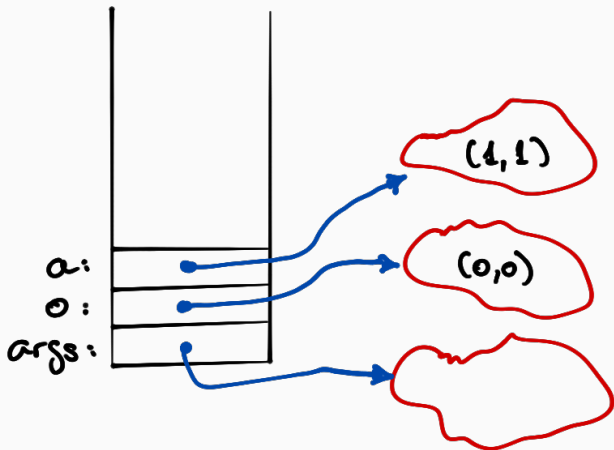
# Pila (*Stack*) de ejecución

Punto2D o = **new** Punto2D()

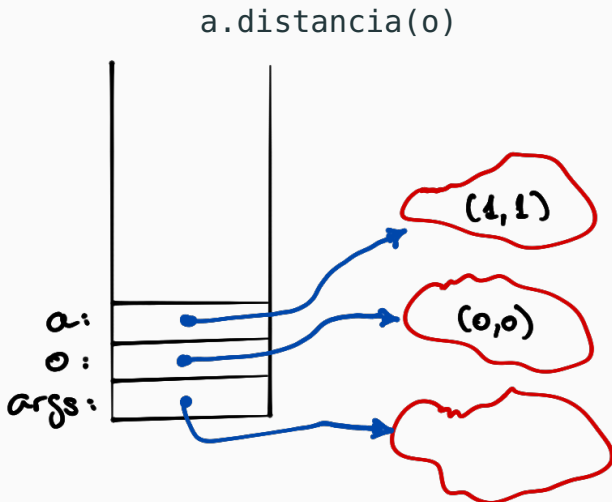


# Pila (*Stack*) de ejecución

```
Punto2D a = new Punto2D(1,1)
```

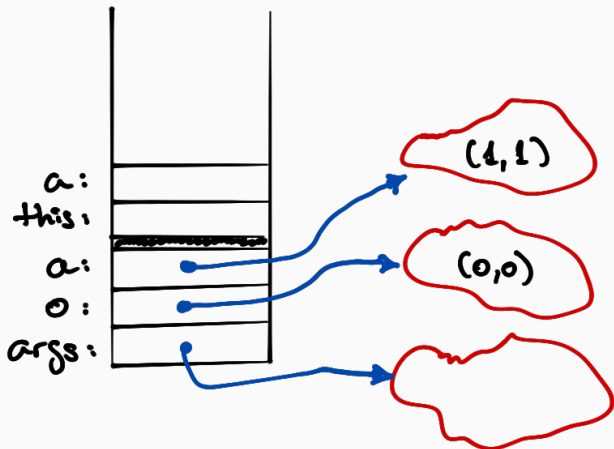


# Pila (*Stack*) de ejecución



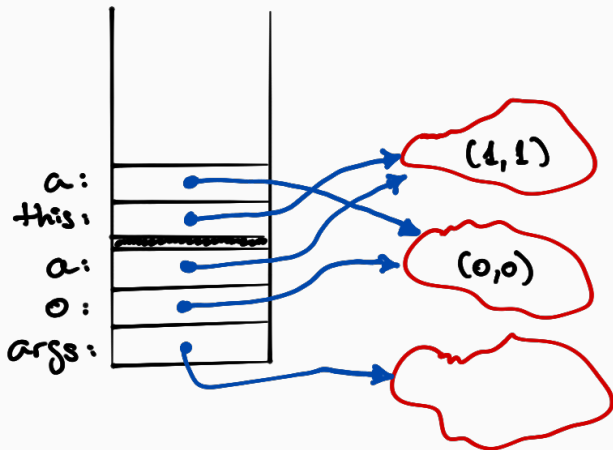
# Pila (*Stack*) de ejecución

`public double distancia(Punto2D a)` 🔔



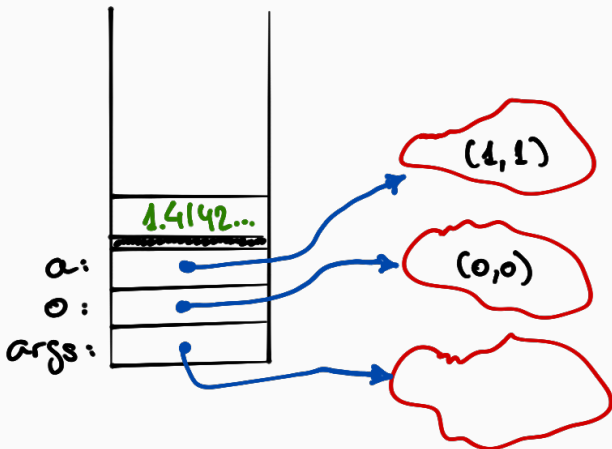
# Pila (*Stack*) de ejecución

`a.distancia(o)`



# Pila (*Stack*) de ejecución

En distancia: **return** Math.sqrt(...);



# Paso de parámetros en Java

- El paso de parámetros en Java se hace *por valor*
- ¿Qué significa *por valor*<sup>1</sup>?

El valor se copia en el argumento

---

<sup>1</sup>También se usa el término *por copia*



# ¿Qué va a pasar?

```
class Intercambiar {  
    public static void  
        main(String args[]) {  
        int x, y;  
        x = 27;  
        y = 42;  
        intercambiar(x,y);  
        System.out.println(  
            x + " - " + y  
        );  
    }  
}
```

a) 27 - 42

b) 42 - 27

# ¿Qué va a pasar?

```
class Intercambiar {  
    public static void  
        main(String args[]) {  
        int x, y;  
        x = 27;  
        y = 42;  
        intercambiar(x,y);  
        System.out.println(  
            x + " - " + y  
        );  
    }  
}
```

a) 27 - 42

b) 42 - 27

 ¡Dibuja!


# ¿Qué va a pasar?

```
class Intercambiar {  
    public static void  
        main(String args[]) {  
        int x, y;  
        x = 27;  
        y = 42;  
        intercambiar(x,y);  
        System.out.println(  
            x + " - " + y  
        );  
    }  
}
```

a) 27 - 42

b) 42 - 27

 ¡Dibuja!

 Programa y ejecuta

# ¿Qué es **static**? i

- Clase como *plantilla*:  
las instancias se crean usando la plantilla
- Cada instancia **tiene** los atributos declarados:  
`p.x`
- Cada instancia **tiene** los métodos declarados.  
`a.distancia(b)`
- Cada instancia **tiene** los suyos y son distintos de los del resto de instancias.

# ¿Qué es **static**? ii

- ¿Y si quiero programar una **función matemática**?

```
public          int fibonacci(int i) {  
    ...  
}
```

- ¿O un *método* que **no dependa de una instancia**?<sup>2</sup>

```
public          void intercambiar(int x, int y) {  
    ...  
}
```

- ¿O un dato *compartido* por todo mi código?

```
public          int MAX_CANCIONES = 1000;}
```

---

<sup>2</sup>¡Recordad que intercambiar no intercambia!

# ¿Qué es **static**? ii

- ¿Y si quiero programar una **función matemática**?

```
public static int fibonacci(int i) {  
    ...  
}
```

- ¿O un *método* que **no dependa de una instancia**?<sup>2</sup>

```
public static void intercambiar(int x, int y) {  
    ...  
}
```

- ¿O un dato *compartido* por todo mi código?

```
public static int MAX_CANCIONES = 1000;
```

---

<sup>2</sup>¡Recordad que intercambiar no intercambia!

# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);
```

# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;  
  
    private Cancion[] lista =  
        new Cancion[MAX_CANCIONES];  
}
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);  
  
        Playlist l = new Playlist();  
        System.out.println(l.MAX_CANCIONES);  
    }  
}
```



# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;  
  
    private Cancion[] lista =  
        new Cancion[MAX_CANCIONES];  
}
```

```
public class Mates {  
    public static  
        int fib(int i) {  
            ...  
        }  
}
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);  
  
        Playlist l = new Playlist();  
        System.out.println(l.MAX_CANCIONES);  
}
```

# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;  
  
    private Cancion[] lista =  
        new Cancion[MAX_CANCIONES];  
}
```

```
public class Mates {  
    public static  
        int fib(int i) {  
            ...  
        }  
}
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);  
  
        Playlist l = new Playlist();  
        System.out.println(l.MAX_CANCIONES);  
  
        System.out.println(Mates.fib(10));  
}
```

# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;  
  
    private Cancion[] lista =  
        new Cancion[MAX_CANCIONES];  
}
```

```
public class Mates {  
    public static  
        int fib(int i) {  
            ...  
        }  
}
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);  
  
        Playlist l = new Playlist();  
        System.out.println(l.MAX_CANCIONES);  
  
        System.out.println(Mates.fib(10));  
  
        // No queremos que compile!  
        Mates = new Mates();  
    }  
}
```

# ¿Qué es **static**? iii

```
public class Playlist {  
    public static  
        int MAX_CANCIONES = 1000;  
  
    private Cancion[] lista =  
        new Cancion[MAX_CANCIONES];  
}
```

```
public class Mates {  
    public static  
        int fib(int i) {  
            ...  
        }  
    private Mates() {  
    }  
}
```

```
public class Principal {  
    public static main(String[] args) {  
        System.out.println(Playlist.MAX_CANCIONES);  
  
        Playlist l = new Playlist();  
        System.out.println(l.MAX_CANCIONES);  
  
        System.out.println(Mates.fib(10));  
  
        // No queremos que compile!  
        Mates = new Mates();  
    }  
}
```

## Tema 2: Colecciones acotadas de objetos

---

Programa que lea de la entrada estándar e imprima en la salida estándar una *playlist*

## Mi *playlist*: entrada

- Lo primero que se lee es un entero que indica cuantas canciones hay
- Por cada canción tres líneas:
  1. Título
  2. Artísta
  3. Valoración (de 0 a 5)

## Mi *playlist*: ejemplo de entrada

3

Despacito

Luis Fonsi

2

The logical song

Supertramp

5

Wish you where here

Pink Floyd

4



## Mi *playlist*: salida

- Cada canción en una línea con este formato:

*Título:Artista:Valoración*

## Mi *playlist*: salida esperada

Despacito:Luis Fonsi:2

The logical song:Supertramp:5

Wish you where here:Pink Floyd:4

## ❓ Para leer de la entrada estándar

- Variable con un `java.util.Scanner`:

```
java.util.Scanner stdin =  
    new java.util.Scanner(System.in);
```

- Leer una línea:

```
String titulo;  
titulo = stdin.nextLine();
```

## ❓ Para leer de la entrada estándar

- Variable con un `java.util.Scanner`:

```
java.util.Scanner stdin =  
    new java.util.Scanner(System.in);
```

- Leer una línea:

```
String titulo;  
titulo = stdin.nextLine();
```

- `java.util.Scanner`

## ❓ Para probar más rápidamente

- Pon toda la entrada en un fichero, por ejemplo `canciones.txt` siguiendo el formato indicado
- Haz que tu programa lea de ese fichero en vez de usar la entrada estándar, para ello utiliza la siguiente línea de comandos:

```
C:\ Sesion07> java ImprimirPlaylist < canciones.txt
Despacito:Luis Fonsi:2
The logical song: Supertramp:5
Wish you where here:Pink Floyd:4
C:\ Sesion07> _
```