

# Programación Orientada a Objetos

# Paradigmas de Programación

- **Imperativos** (énfasis en la ejecución de instrucciones)
  - Programación Procedimental (ej. Pascal)
  - Programación Orientada a Objetos (ej. Smalltalk)
- **Declarativos** (énfasis en la evaluación de expresiones)
  - Programación Funcional (ej. Haskell)
  - Programación Lógica (ej. Prolog)

Los lenguajes más utilizados (por ejemplo, Java) son **multiparadigma**. Cabe a los programadores usar el estilo de programación más adecuado para cada trabajo.

# OOP: Historia

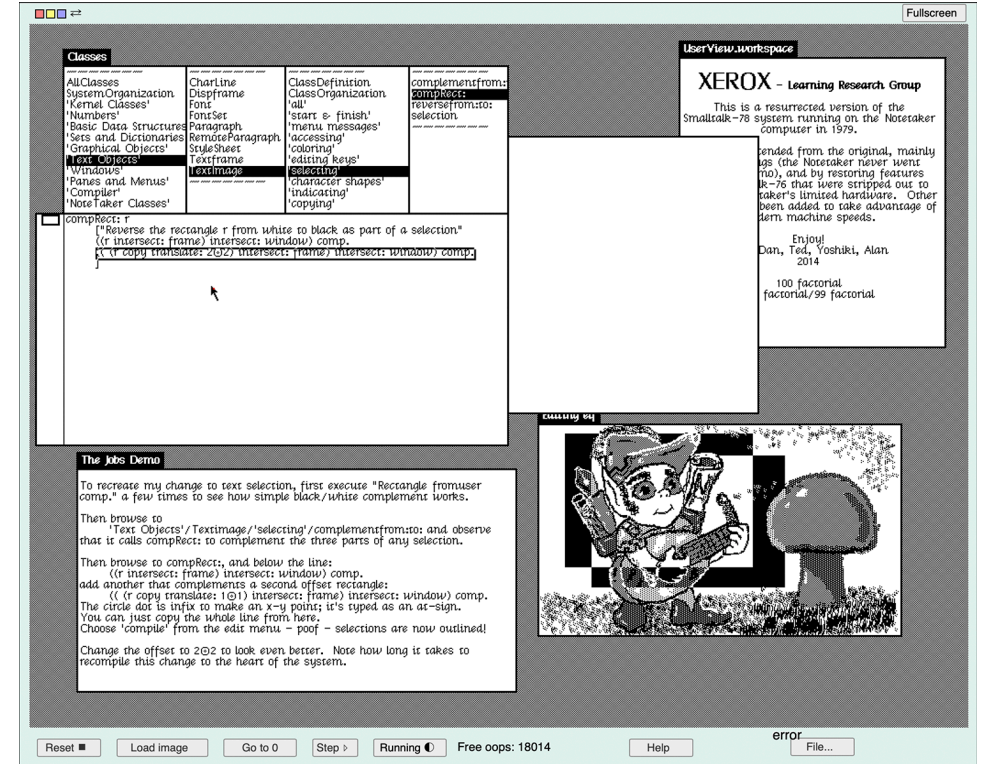
**1950s:** La idea de “objetos” surge del grupo de IA del MIT

**1960s:** Simula: primer language con objetos, clases y herencia

**1970s:** Smalltalk (Alan Kay): Primer lenguaje OOP puro

**1980s:** OOP gana popularidad

- C++ (Bjarne Stroustrup)
- Objective-C (Brad Cox → NeXT → Apple)
- Eiffel (Bertrand Meyer)
- Delphi (Object Pascal, Anders Hejlsberg → Borland)



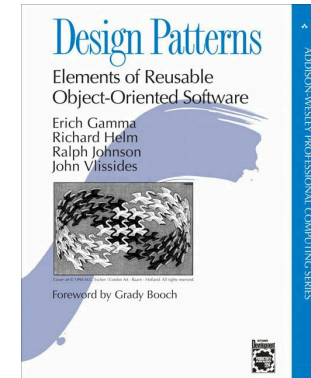
# OOP: Historia (cont.)

## 1990s-2000s: OOP es mainstream

- Java (James Gosling → Sun Microsystems → Oracle)
- Python (Guido van Rossum)
- Ruby (Yukihiro Matsumoto)
- .NET Framework, C#, VB.NET (Microsoft)
- Design patterns (GoF)
- UML

## 2010s-actualidad: Scala, Kotlin, Swift, Go, Rust, ...

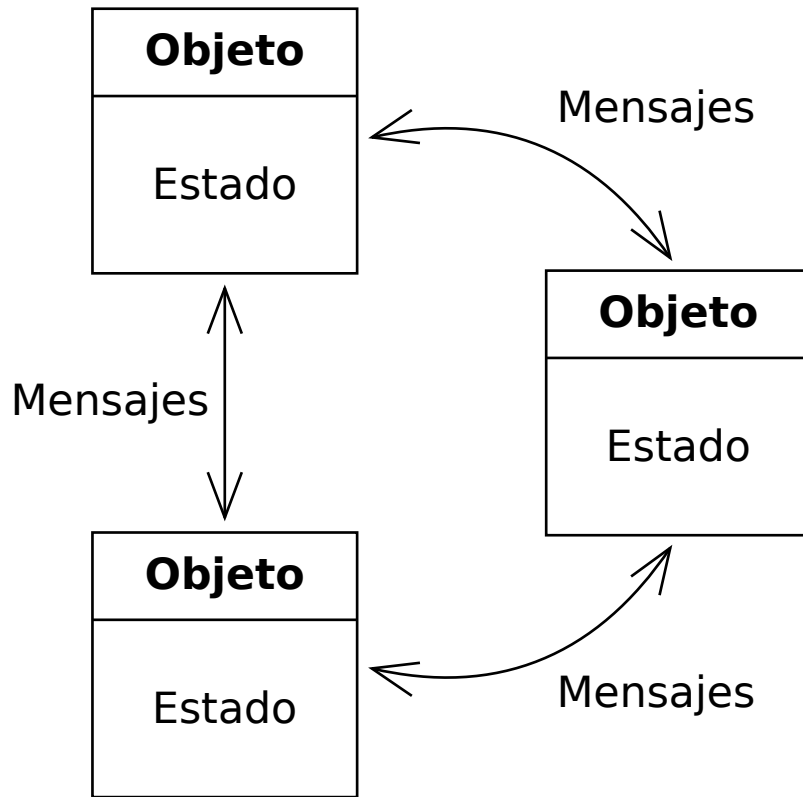
- Los lenguajes modernos tienden a combinar características de OOP y FP



# Objetos

Un **objeto** es una entidad (habitualmente almacenada en memoria) que tiene **identidad**, **estado** y **comportamiento**.

Un sistema orientado a objetos se forma de un conjunto de objetos que interactúan pasando **mensajes** entre sí.



# Estilos de OOP

Una característica importante en los lenguajes orientados a objetos es la posibilidad de definir objetos con comportamiento similar. Hay dos variantes de lenguajes según cómo se implementa esto:

**Basado en Clases:** Todo objeto es una **instancia** de una **clase** específica.

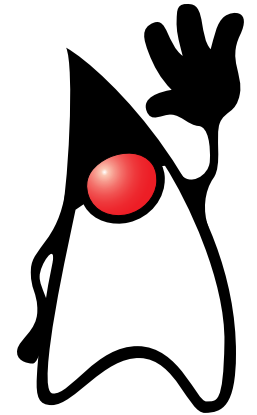
Ejemplos: Smalltalk, Java, C++, C#, Python, Ruby, Swift, Kotlin

**Basado en Prototipos:** Todo objeto está asociado a otro (su **prototipo** o **padre**).

Ejemplos: JavaScript, Lua

# Java

- Creado por James Gosling en 1995 en Sun Microsystems
- De propósito general
- Multiparadigma, principalmente OOP
- Basado en clases
- Recolector de basura automático
- Tipado estático y fuerte
- Compilación a bytecode y ejecución en una máquina virtual (JVM)
  - Portabilidad: “Write once, run anywhere”
  - Windows, Linux, macOS, Android, etc.



# Definiciones

**Clase:** Plantilla para crear objetos. Define los aspectos que comparten todos los objetos creados a partir de la clase (**atributos** y **métodos**).

**Instancia:** Un objeto creado a partir de una clase. Cada instancia tiene su propia **identidad**, que permite distinguirla de otras instancias. Al crearse una instancia se reserva memoria para almacenar su **estado**.

**Atributos:** Variables de instancia que almacenan el **estado** de un objeto. Alias: **miembros**, **campos**.

**Métodos:** Funciones que operan sobre un objeto. El conjunto de métodos definidos por una clase determinan el **comportamiento** del objeto. La forma de enviar un **mensaje** a un objeto es invocar uno de sus métodos.

**Constructor:** Método especial que es invocado automáticamente cuando se crea una instancia de la clase.

```
class Punto {  
    double x;  
    double y;  
  
    Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    double norma() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

```
Punto p = new Punto(2, 5);  
System.out.println(p.norma()); // 5.3851
```



# Ejemplo: Número secreto

```
public class Juego {  
    private final int numeroSecreto;  
    private int intentos;  
  
    public Juego(int maxNumero, int maxIntentos) {  
        this.intentos = maxIntentos;  
        Random rand = new Random();  
        this.numeroSecreto = rand.nextInt(maxNumero) + 1;  
    }  
  
    public int getIntentos() {  
        return intentos;  
    }  
  
    public Resultado intentar(int n) {  
        assert this.intentos > 0;  
        this.intentos--;  
  
        if (n < this.numeroSecreto) {  
            return new Resultado(false, "Muy chico!");  
        } else if (n > this.numeroSecreto) {  
            return new Resultado(false, "Muy grande!");  
        } else {  
            return new Resultado(true, "Acertaste!");  
        }  
    }  
}
```

```
public class Resultado {  
    public final boolean esCorrecto;  
    public final String mensaje;  
  
    public Resultado(boolean esCorrecto, String mensaje) {  
        this.esCorrecto = esCorrecto;  
        this.mensaje = mensaje;  
    }  
}
```

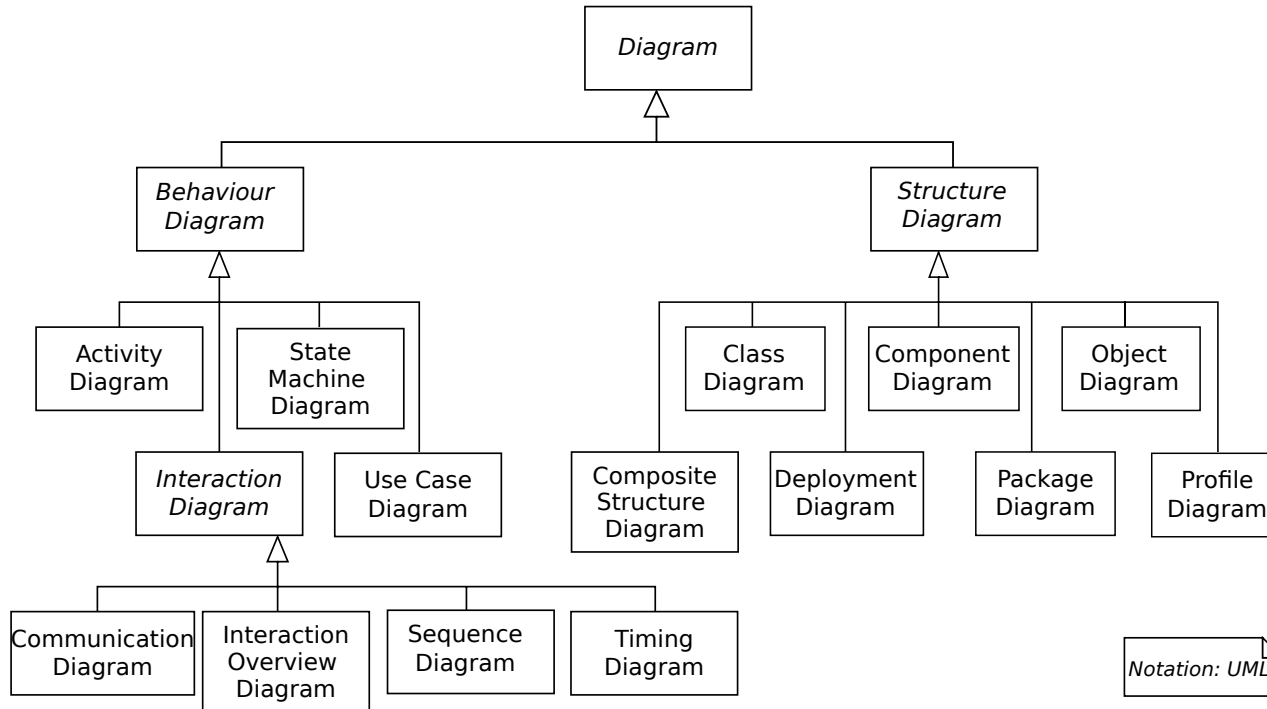
# Ejemplo: Número secreto (cont.)

```
public class JuegoApp {  
    public static final int MAX_NUMERO = 100;  
    public static final int MAX_INTENTOS = 5;  
  
    private final Scanner scanner;  
    private final Juego juego;  
  
    public JuegoApp(Scanner scanner) {  
        this.scanner = scanner;  
        this.juego = new Juego(MAX_NUMERO, MAX_INTENTOS);  
    }  
}
```

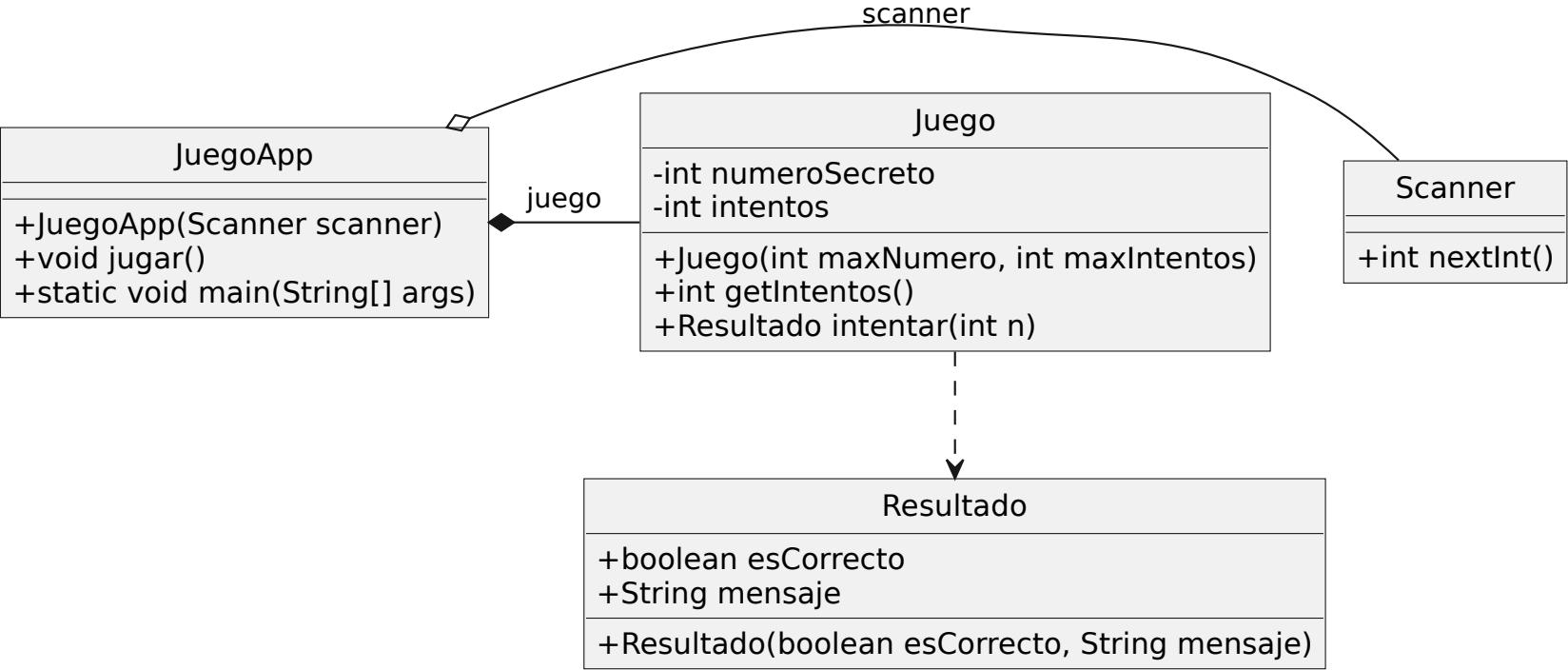
```
    public void jugar() {  
        System.out.printf(  
            "Adivina el número secreto entre 1 and %d." +  
            "Tienes %d intentos.%n\n",  
            MAX_NUMERO,  
            MAX_INTENTOS  
        );  
        while (juego.getIntentos() > 0) {  
            System.out.printf("Quedan %d intentos.\n", juego.getIntentos());  
            System.out.print("Adivina el numero: ");  
            int n = scanner.nextInt();  
  
            Resultado r = juego.intentar(n);  
            System.out.println(r.mensaje);  
            if (r.esCorrecto) {  
                break;  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        JuegoApp app = new JuegoApp(scanner);  
        app.jugar();  
    }  
}
```

# UML

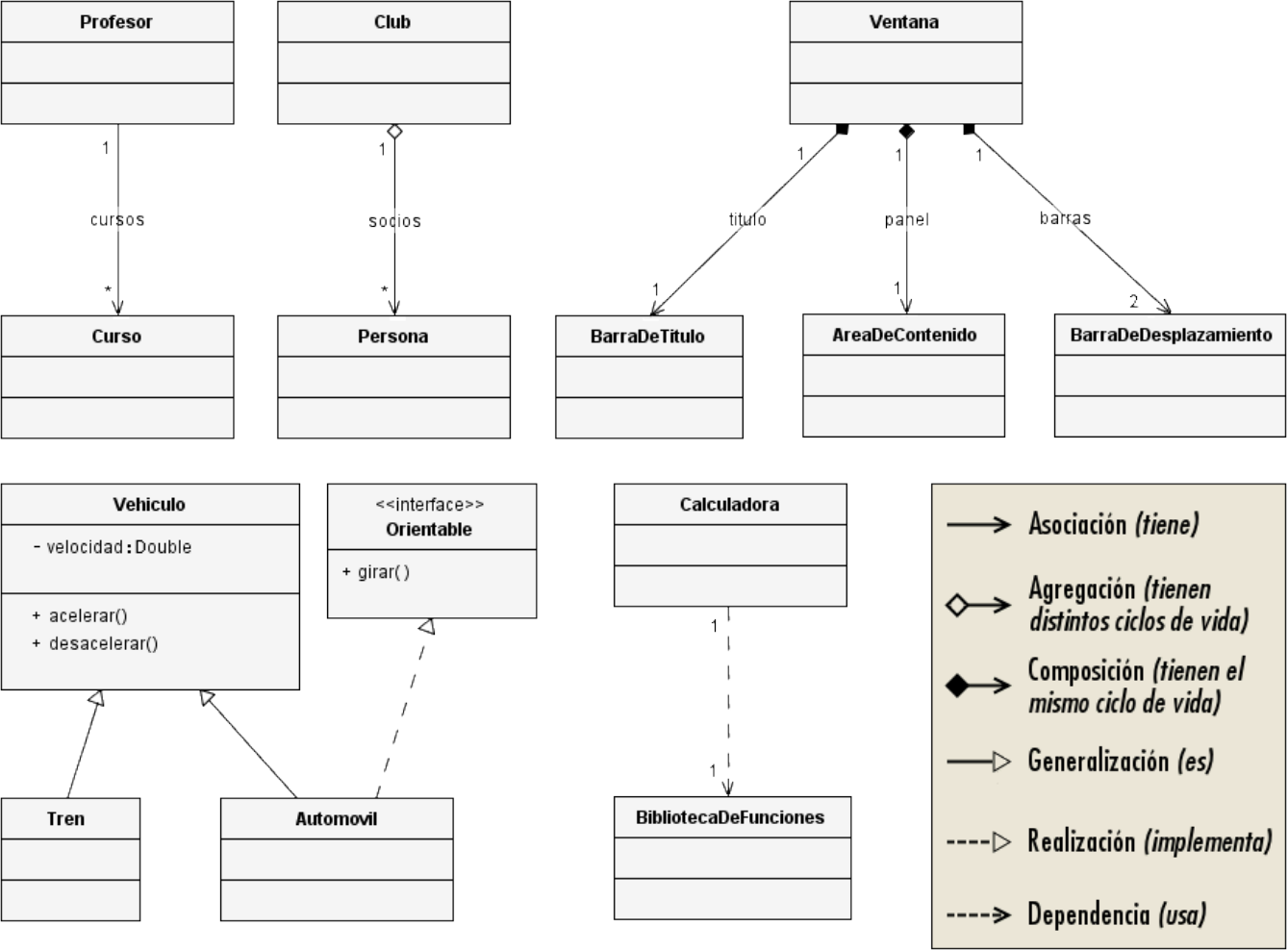
**Unified Modeling Language:** proporciona una forma estándar de visualizar el diseño de un sistema.



# UML: Diagrama de clases

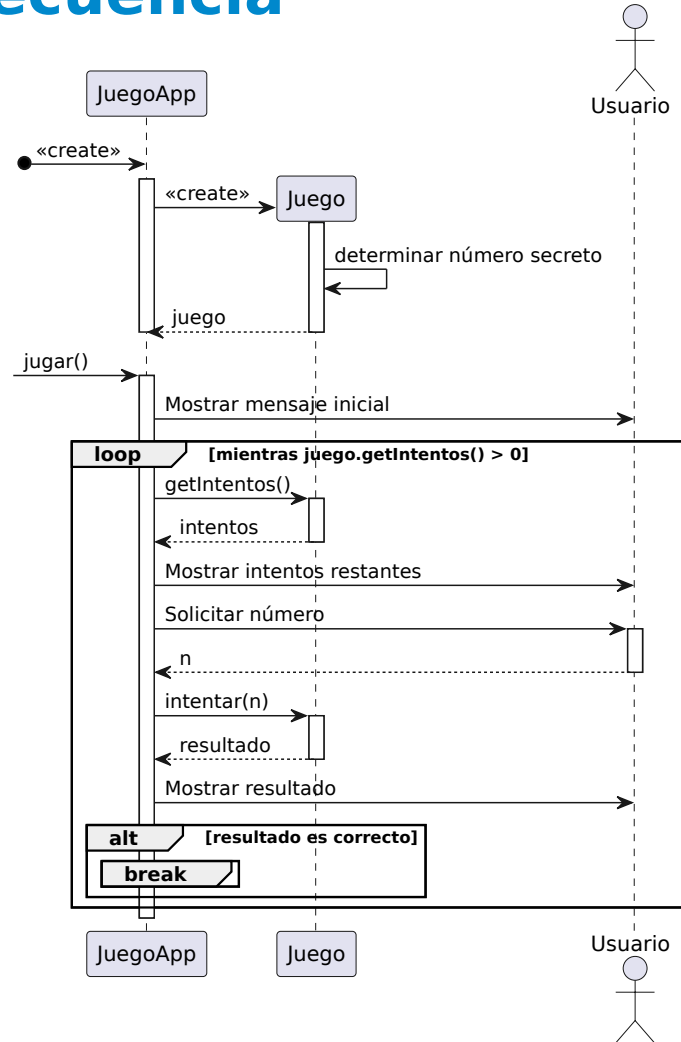


# Relaciones entre clases (e interfaces)



- |      |               |  |
|------|---------------|--|
| 1    | uno           | } <b>Multiplicidad</b><br>(Número de elementos relacionados) |
| 0..1 | cero o uno    |  |
| 0..* | cero o muchos |  |
| 1..* | uno o muchos  |  |
| *    | cero o muchos |  |

# UML: Diagrama de secuencia



# Java: Generalidades

- El código se organiza en **paquetes** (carpetas) y **clases** (usualmente un archivo `.java` por cada clase).
- Guía de estilo: <https://github.com/pepperkit/java-style-guide>
  - Clases: `PascalCase` (ej. `Punto`)
  - Variables, atributos, y métodos: `camelCase` (ej. `x`, `calcularNorma`)
  - Constantes: `UPPER_CASE` (ej. `PI`, `MAX_INT`)
- Crear una instancia de una `Clase`: `new Clase(...)`
- Acceder a un atributo, invocar un método: `instancia.atributo`, `instancia.metodo(...)`
- En un objeto mutable, es recomendable calificar sus atributos como `private`, para evitar que puedan ser manipulados desde fuera de la clase, y opcionalmente definir **métodos de acceso** (*getters* y *setters*) públicos.
- El modificador `final` indica que una variable o atributo es inmutable.
- El modificador `static` indica que un atributo o método pertenece a la clase en sí, y no a una instancia específica.

# Tipos de datos

- Dos categorías: [\[#\]](#)
  - **Primitivos:** `int`, `double`, `boolean`, `char`, etc. (valor por defecto: `0`)
  - **Referencias:** apuntan a un objeto en memoria (ej: `Punto`, valor por defecto: `null`)
- Arreglos: [\[#\]](#)
  - Declaración: `int[] numeros;`
  - Inicialización: `numeros = new int[10];`    `numeros = {1, 2, 3};`
  - Acceso: `numeros[0]`    `numeros.length`
  - El tamaño de un arreglo es fijo una vez creado
- Strings: [\[#\]](#)
  - Son instancias de la clase `String`
  - `"..."` es azúcar sintáctica para `new String(...)`
  - Inmutables
- Colecciones dinámicas: [\[#\]](#)
  - Clases disponibles en la biblioteca estándar
  - `ArrayList`, `HashSet`, `HashMap`, etc.



**[www.ingenieria.uba.ar](http://www.ingenieria.uba.ar)**

**f**    /ingenieriauba

 /FIUBAoficial