

# Interfaces y Clases Internas

(No es interfaz gráfica, sirve como alternativa a herencia múltiple)

ELO329: Diseño y Programación Orientados a Objetos

# ¿Qué queremos decir con interfaces y clases internas?

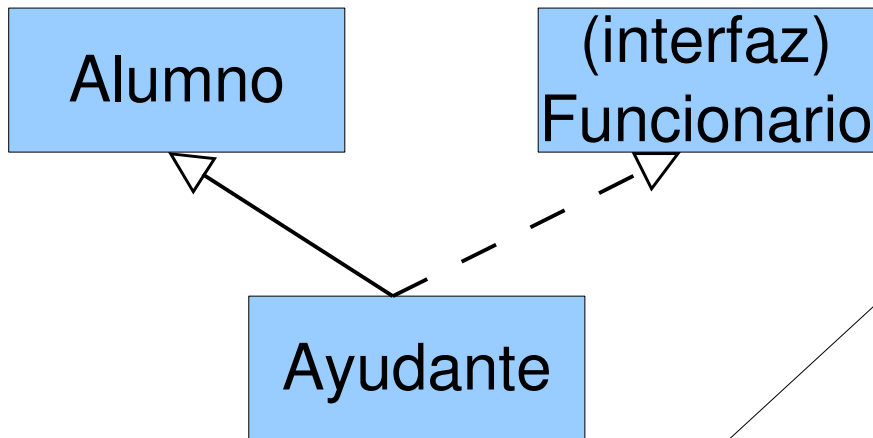
- El término interfaz aquí **NO** se refiere a las interfaces gráficas (esas ya vienen ...).
- Aquí las **interfaces** son una manera de describir qué debería hacer una clase sin especificar el cómo.
- Faz<=> cara; “entre caras” => interfaz, plural interfaces
- Las **clases internas** son clases anidadas dentro de otras clases o métodos.
- Entre otros, interfaces y clases internas son recursos esenciales en el manejo de **interfaces gráficas** en Java. Será nuestro próximo tópico.

# Interfaces

- Una **interfaz** es la descripción de uno o más servicios (métodos) que posteriormente alguna clase puede implementar (y por ende ofrecer).
- Por ejemplo, si un alumno es ayudante, entonces podríamos preguntarle por su sueldo u otras cosas propias de un funcionario. Es así como un ayudante además de ser alumno es capaz de responder consultas propias de un funcionario. Así un Ayudante además de ser Alumno (herencia) cumple con la interfaz Funcionario. También podríamos decir que él **es un** “Funcionario” (la misma relación que en herencia).
- **Gracias a las interfaces podemos crear métodos genéricos (como ordenar) que manipulen objetos y les exijan a éstos sólo funcionalidades mínimas requeridas (como poder compararse).**

# Representación gráfica

Ejemplo 1



Ejemplo 2

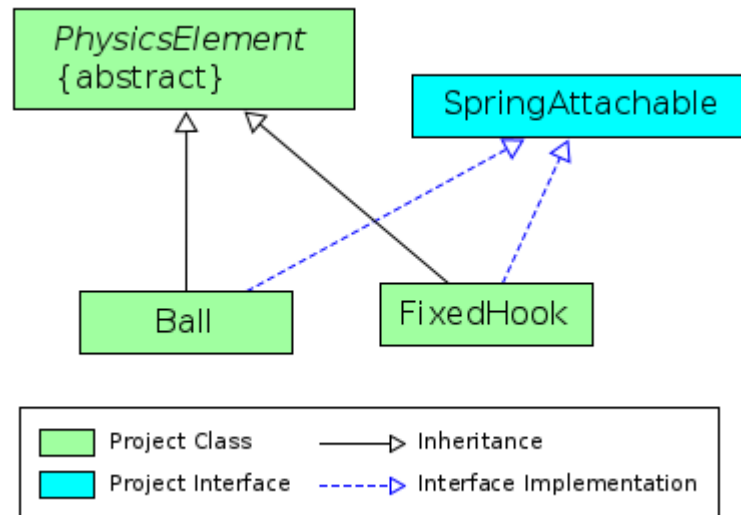


Diagrama generado por  
Jgrasp

# Interfaces: aspectos operativos

1° Paso

Definir la Interfaz:  
Incluir todos los métodos que objetos deben ser capaces de ofrecer.

Como resultado tenemos un archivo que define la interfaz  
Ej. Funcionario.java

2° Paso

Implementación es hecha en alguna clase

Como resultado tenemos una clase que implementa todos los métodos de la interfaz Funcionario, además de los propios de la clase.  
Ej. clase Ayudante

Ej. Implementación de Ayudante

Métodos de Funcionario

## Interfaces (cont.)

- En Java cada clase puede tener sólo una clase base (en Java no hay herencia múltiple).
- Cuando hay relación **es-un** con más categorías del mundo real, usamos herencia con una de ellas e interfaces para exhibir el comportamiento esperado para las otras.
- Se **cumple también el principio de sustitución**.  
Instancias de la clase que **implementa** una Interfaz pueden ser usadas donde hay una referencia a una instancia de la interfaz. Es similar a usar una instancia de una subclase cuando se espera un objeto de la clase base.

# Interfaces (cont.)

- No se permite crear instancias (objetos) de una Interfaz. Por la misma razón que no se puede crear instancias de clases abstractas, no se tienen la implementaciones.

`new InterfazX();` ✗

- Todos los métodos de una Interfaz son públicos. No es necesario indicarlo.
- Pueden incluir constantes. En este caso son siempre `public static final`.

# Interfaces: Aspectos sintácticos

- Debemos atender dos cosas:
  - Si la interfaz no existe en el lenguaje, la debemos definir.

**Definición** de una interfaz, en un archivo de nombre Comparable.java, poner:

```
public interface Comparable {  
    int compareTo (Object other);  
}
```

- Luego debemos implementar la interfaz en alguna clase.

**Implementación** de una interfaz:

```
class Employee implements Comparable {  
    ....  
    public int compareTo(Object other) {  
        ....// implementación  
    }  
}
```



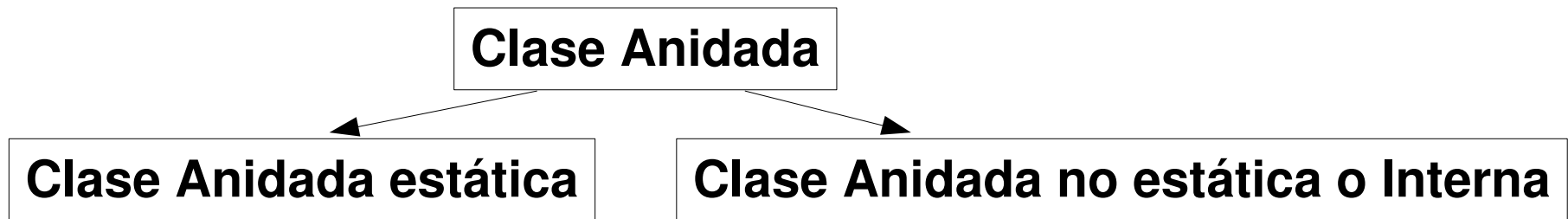
# Ejemplo: uso de interfaces

- Consideremos la extensión de la clase Employee para que podamos ordenar arreglos de empleados según su salario.
- La interfaz Comparable ya está definida en el lenguaje, luego sólo debemos implementarla.
- Ver EmployeeSortTest.java
- Ver documentación de clase Arrays e interfaz Comparable. Notar el métodos genérico sort de la clase Arrays.

# Clases Anidadas

- Clases anidadas son clases definidas al interior de otra clase.
- Tres razones para ofrecer esto en Java:
  - Es una manera de agrupar clases usadas en un lugar.
  - Aumenta la encapsulación
  - Generan código más legible y mantenible.
- Cuando sólo necesitamos crear instancias de una clase al interior de sólo una clase, podemos definir la clase al interior de ésta.
- Pueden haber clases anidadas estáticas o no estáticas, a estas últimas se les llama internas.

# Clases Anidadas: hay de dos tipos



- Ejemplo:

```
class OuterClass { // Clase anfitriona
    ...
    static class StaticNestedClass { // estática
        ...
    }
    class InnerClass { // interna
        ...
    }
}
```

# Clases Anidadas (cont.)

- Las clases internas tienen acceso a los atributos de la anfitriona o externa (incluso si son privados) pero las anidadas estáticas no.
- Las estáticas **están asociadas a la clase** anfitriona.
- Las internas **están asociadas a instancias** de clase anfitriona.
- Como miembros de la clase anfitriona, las clases anidadas pueden ser declaradas private, public, protected o privadas del paquete (cuando omitimos el calificador).
- Son muy útiles para reducir código fuente. Especialmente cuando la clase sólo genera instancias locales.
- Son comunes en el desarrollo de interfaces gráficas.

# Clases Anidadas (Cont.)

- Las clases internas existen sólo para el compilador, ya que éste las transforma en clases regulares separando la clase externa e interna con signo \$.
- La máquina virtual no distingue la clases internas.
- También se pueden definir al interior de un método.
- Ejemplo creación de una instancia (en declaraciones public):
  - Clases estáticas anidadas:

```
OuterClass.StaticNestedClass nestedObject =  
                                new OuterClass.StaticNestedClass();
```
  - Clase interna (no estática): **OJO primero debemos crear un objeto:**

```
OuterClass outerObject = new OuterClass();  
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```
- En este curso usaremos principalmente clases anidadas no públicas.

# Ejemplo de Clase Interna

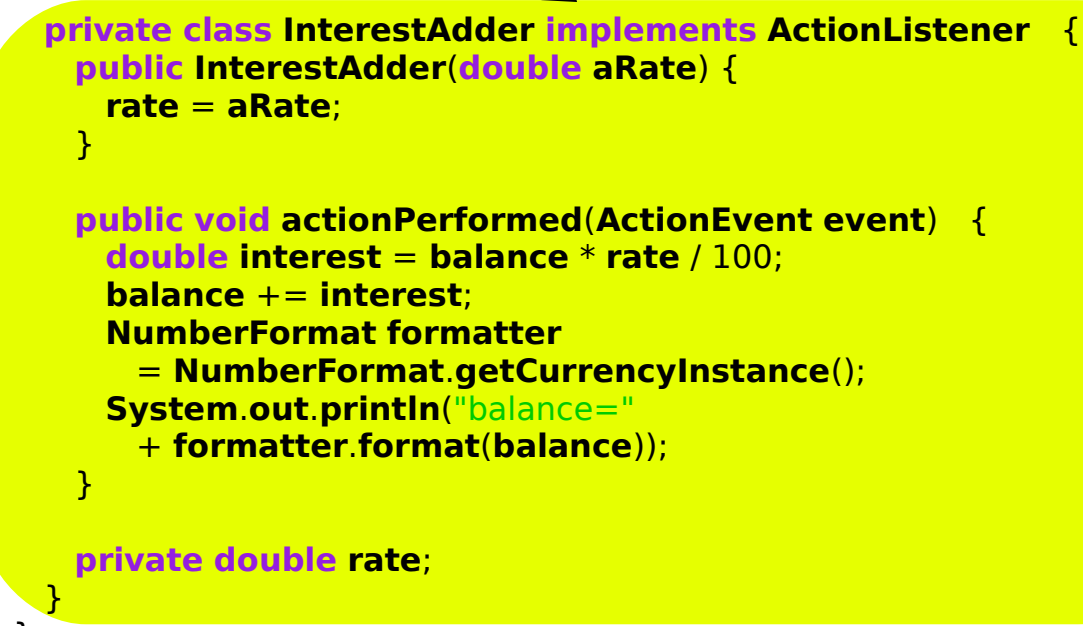
```
class BankAccount {
    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    public void start(double rate)
    {
        ActionListener adder = new InterestAdder(rate);
        Timer t = new Timer(1000, adder);
        t.start();
    }
    private double balance;

    private class InterestAdder implements ActionListener {
        public InterestAdder(double aRate) {
            rate = aRate;
        }

        public void actionPerformed(ActionEvent event) {
            double interest = balance * rate / 100;
            balance += interest;
            NumberFormat formatter
                = NumberFormat.getCurrencyInstance();
            System.out.println("balance="
                + formatter.format(balance));
        }

        private double rate;
    }
}
```



Ver InnerClassTest.java

# Clase interna dentro de un método

```
class BankAccount {  
    public BankAccount(double initialBalance) {  
        balance = initialBalance;  
    }  
    public void start(double rate) {
```

```
class InterestAdder implements ActionListener {  
    public InterestAdder(double aRate) {  
        rate = aRate;  
    }  
  
    public void actionPerformed(ActionEvent event) {  
        double interest = balance * rate / 100;  
        balance += interest;  
        NumberFormat formatter = NumberFormat.getCurrencyInstance();  
        System.out.println("balance=" + formatter.format(balance));  
    }  
    private double rate;  
}
```

```
    ActionListener adder = new InterestAdder(rate);  
    Timer t = new Timer(1000, adder);  
    t.start();  
}  
  
private double balance;  
}
```

Ver:  
InnerClassMethodTest.java

# Clases internas anónimas

- ¿Si sólo deseáramos proveer de una implementación a los métodos de una interfaz, para qué definir una clase?
- Cuando necesitamos sólo una instancia de una clase que implementa una interfaz, no necesitamos darle un nombre. Decimos que tal clase es interna y anónima.
- Ver `AnonymousInnerClassTest.java`



# Ejemplo: Clase Anónima

```
class BankAccount
```

```
{
```

```
    public BankAccount(double initialBalance) {  
        balance = initialBalance;
```

```
    }
```

```
    public void start(final double rate) { // adder es Única instancia
```

```
        ActionListener adder = new ActionListener() { ← // Implementación
```

```
            public void actionPerformed(ActionEvent event) {
```

```
                double interest = balance * rate / 100;
```

```
                balance += interest;
```

```
                NumberFormat formatter = NumberFormat.getCurrencyInstance();
```

```
                System.out.println("balance=" + formatter.format(balance));
```

```
            }
```

```
        };
```

```
        Timer t = new Timer(1000, adder);
```

```
        t.start();
```

```
    }
```

//Ver

```
    private double balance;
```

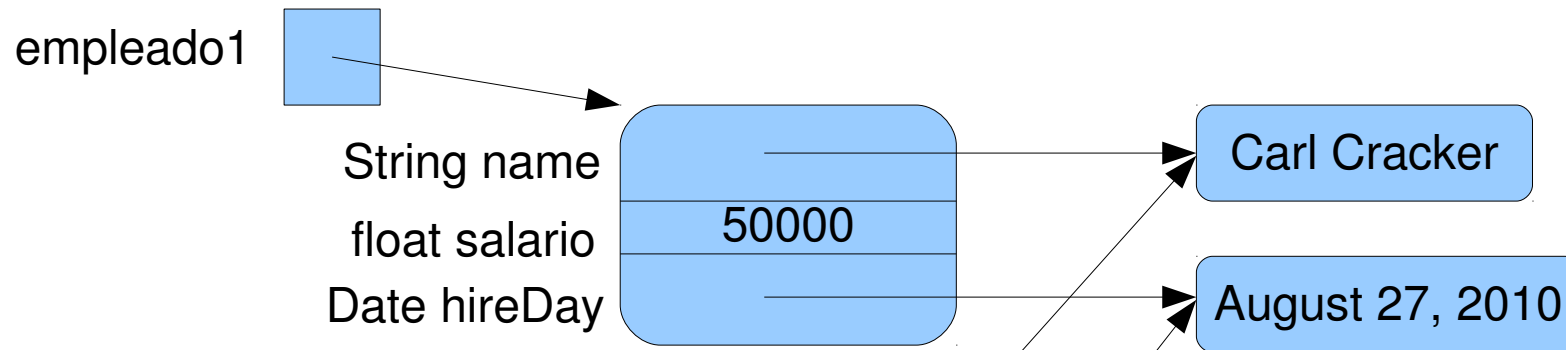
AnonymousInnerClassTest.java

```
}
```

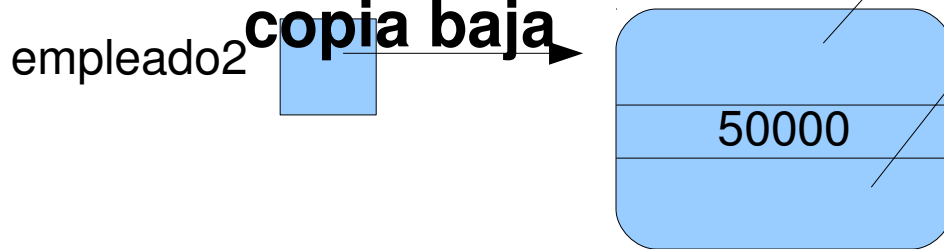
# Método clone() en Object (revisitado)

- El método clone() existe con acceso protegido en la clase Object.
- Para invocarlo sobre un objeto se requiere que la clase del objeto implemente la interfaz Cloneable, lo cual significa que debemos redefinir el método clone.
- Para generar un clone correcto, debemos hacerlo invocando el método clone de la clase Object.
- El método clone de Object crea y retorna un objeto con igual estructura del objeto llamado e inicializa todos sus campos con el mismo contenido de los campos del objeto llamado.
- Los contenidos de cada campo no son clonados (hasta aquí se le llama copia baja), luego para una copia completa (profunda) se debe llamar el método clone con cada atributo.

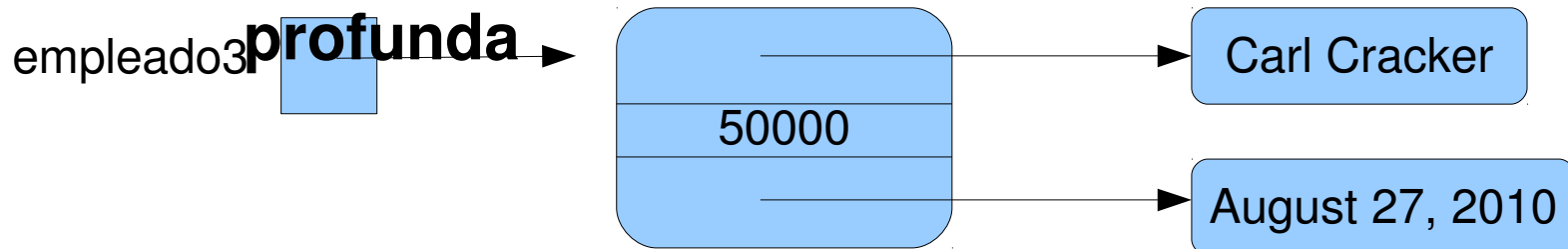
# Copia baja v/s copia profunda



**empleado2 = empleado1.clone(); // si fuera como  
copia baja**



**empleado3 = empleado1.clone(); // si fuera copia  
profunda**



# Implementación de clone (copia profunda)

- La implementación típica es como sigue:

```
class Employee implements Cloneable {  
    public Object clone() { // redefinición de clone  
        try { // el manejo de excepciones de revisará más adelante  
            Employee c = (Employee) super.clone(); // no usamos constructor  
            c.hireDay = hireDay.clone();  
            return c;  
        } catch (CloneNotSupportedException e ) {  
            return null;  
        }  
    }  
    .....  
    private String name;  
    private float salary;  
    private Date hireDay;  
}
```

Hasta aquí copia baja

Necesarios para copia profunda

String es no mutante, no requerimos clonarlo

Ver CloneTest.java