Interfaces y Clases Anidadas

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

ELO329: Diseño y Programación Orientados a Objetos

¿Qué queremos decir con interfaces y clases anidadas?

- El término interfaz aquí **NO** se refiere a las interfaces gráficas (esas ya vienen …)
- Una interfaz, como una clase, declara el comportamiento esperado para objetos; sin embargo, a diferencia de una clase, no se especifica el cómo, es decir, la implementación de cada método.
- Las clases anidadas son clases definidas dentro de otras clases o métodos.
- Entre otros, interfaces y clases internas son recursos esenciales en el manejo de *interfaces gráficas* en Java. Éste será el próximo tópico.

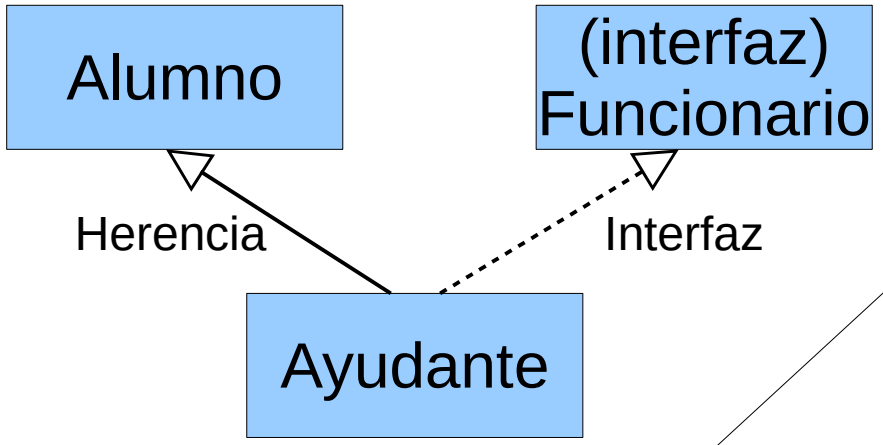
Nota lingüística: Faz<=> cara; “entre caras” => interfaz, plural interfaces

Interfaces

- Una interfaz es la descripción de uno o más servicios o comportamientos (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 solo funcionalidades mínimas requeridas (como poder compararse).

Representación gráfica UML

Ejemplo 1



Ejemplo 2

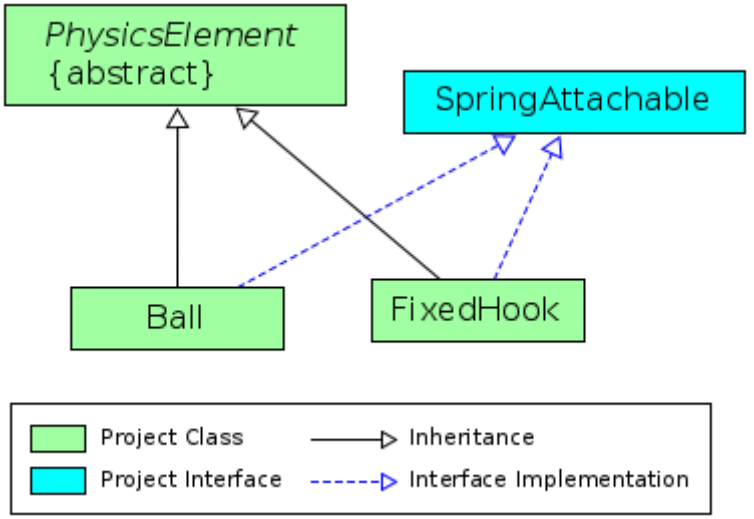


Diagrama generado por Jgrasp

Interfaces: aspectos operativos

1° Paso
Definir la Interfaz:
Incluir solo los prototipos de los métodos de la interfaz.

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

2° Paso
Implementación de la Interfaz: ésta debe ser hecha en alguna clase. Para cada método, ponemos su prototipo y su implementación

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

Interfaces: Ejemplo 1

- Definición de la interfaz

```
public interface Funcionario {
    int getSalary();
}
```

Java ya predefine algunas Interfaces, ej. Comparable

- Implementación de la interfaz

```
class Ayudante extends Estudiante implements Funcionario {
  public Ayudante(String name, String major, int s) {
     super(name, major);
     salary =s;
  }
  public int getSalary() {   // aquí implementamos la interfaz
     return salay;           // podrían haber varios métodos
  }
  private int salary;
}
```

Interfaces: Ejemplo 2

- 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
 }
}
```

Interfaces (cont.)

- En Java cada clase puede tener sólo una clase base (en Java no hay herencia múltiple), pero cada clase puede implementar varias interfaces.
- Cuando hay relación es-un con varias 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, dado que no se tienen la implementaciones.

  new InterfazX();  // Error!
- Todos los métodos de una Interfaz son públicos. No es necesario indicarlo.
- Pueden incluir constantes. En este caso son siempre public static final.

Ejemplo práctico: uso de interfaces

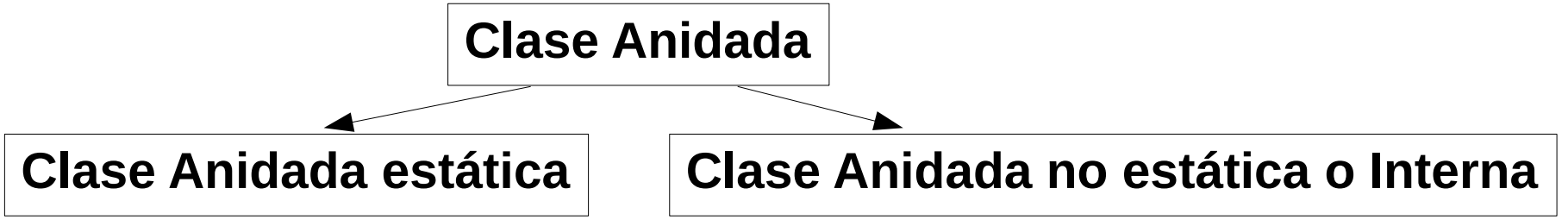
- Consideremos la extensión de la clase Employee para que podamos ordenar arreglos de empleados según su salario.
- La interfaz Comparable ya está definida en el lenguaje, luego solo debemos implementarla.
- Ver EmployeeSortTest.java
- Ver documentación de clase Arrays e interfaz Comparable. Notar el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

Clases Anidadas

Clases Anidadas

- 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 una clase.
  - Aumenta la encapsulación
  - Generan código más legible y mantenible.
- Cuando usamos instancias de una clase B solo al interior de una clase A, podemos definir B al interior de A.
- Pueden haber clases anidadas estáticas y no estáticas, a estas últimas se les llama también clases internas.

Clases Anidadas: hay de dos tipos



- Ejemplo:

```
class OuterClass {  // Clase anfitriona
    ...
        static class StaticNestedClass { // estática
            ...
        }
        class InnerClass {  // no estática o interna
            ...
        }
}
```

Clases Anidadas (cont.)

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

Clases Anidadas (Cont.)

- Las clases anidadas existen sólo para el compilador, ya que éste las transforma en clases regulares separando la clase externa y anidada con signo $.
- La máquina virtual no distingue clases anidadas o de no anidadas.
- 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();
    ```

  - Clases internas (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();        // invoca a método de Timer
  }
  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;      // notar que tiene acceso a balance
      NumberFormat formatter = NumberFormat.getCurrencyInstance();
      System.out.println("balance=" + formatter.format(balance));
    }

    private double rate;
  }
}
```

Instancias de la clase Timer un vez iniciados invocan al método actionPerformed regularmente.

Ver InnerClassTest.java

Solo ocupamos una instancia de la clase interna en método start

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();
  }  // fin del método start

  private double balance;
}
```

Ver:
InnerClassMethodTest.java

Clases internas anónimas

- Si deseáramos proveer de una implementación a los métodos de una interfaz para crear un único objeto ¿para qué definir una clase?
- Cuando necesitamos solo una instancia de una clase que implementa una interfaz, no necesitamos darle un nombre. Decimos que tal clase es interna y anónima.

Ejemplo: Clase Anónima

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

  public void start(final double rate)   {
    ActionListener adder = new   ActionListener()  {
        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();
  }

  private double balance;
}
```

// adder es Única instancia

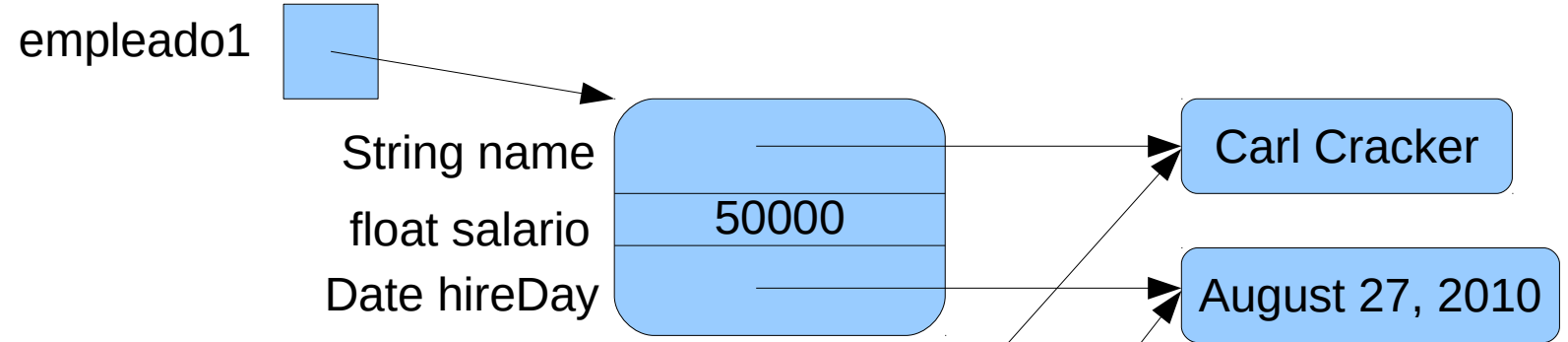
// Implementación

//Ver AnonymousInnerClassTest.java

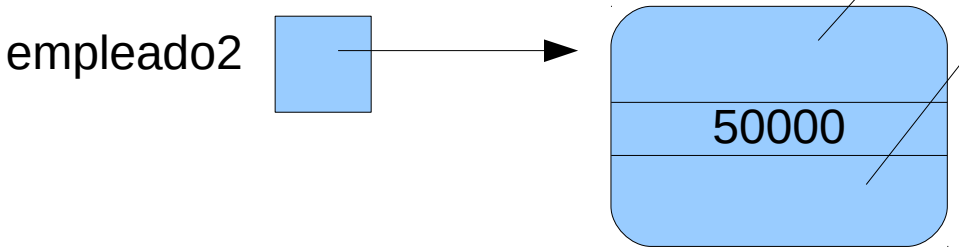
Método clone() en Object (revisitado)

- 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 al 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 de cada atributo.

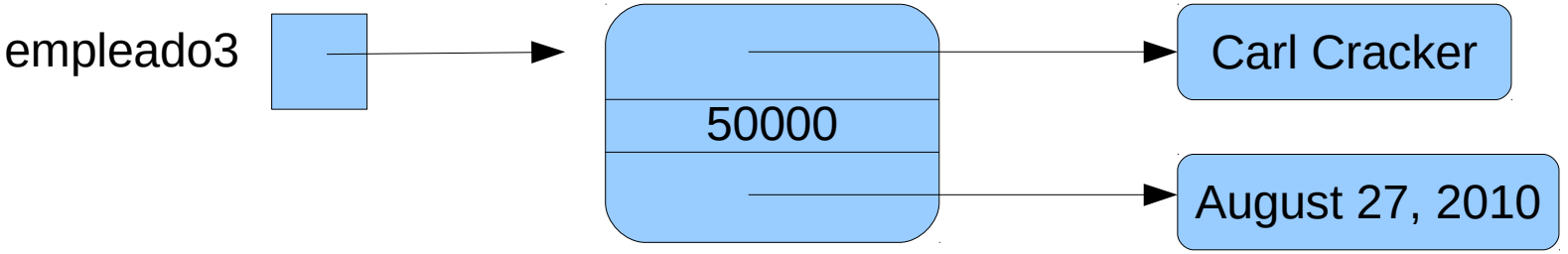
Copia baja v/s copia profunda



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 se 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