Herencia y Clases Abstractas

ELO329: Diseño y Programación Orientados a Objetos

Introducción

- La idea básica es poder crear clases basadas en clases ya existentes.
- Cuando heredamos de una clase existente, estamos re-usando código (métodos y atributos).
- Podemos agregar métodos y variables para adaptar la clase a la nueva situación.
- Java también permite consultar por la estructura de una clase (cuáles son sus métodos y variables). A esto se le llama reflexión (los animales sabrán que son animales, el hombre tiene conciencia sobre su existencia, Java puede consultar por la naturaleza de cada objeto ...)

Introducción (cont.)

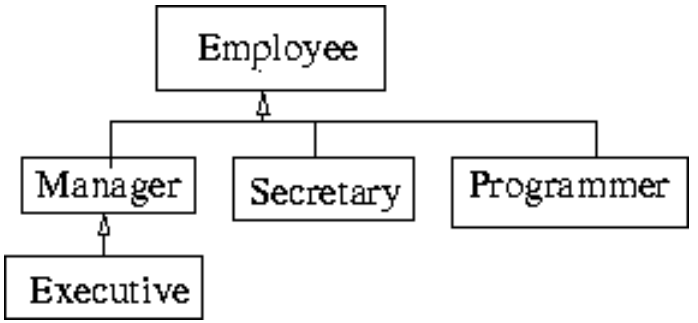
- La herencia la identificamos cuando encontramos la relación es-un entre la nueva clase y la ya existente. Un estudiante es una persona.
- La relación es-un es una condición necesaria pero so suficiente, además los objetos de la clase heredada deben cumplir el principio de sustitución.
- La clase ya existente se le llama superclase, clase base , o clase padre.
- A la nueva clase se le llama subclase, clase derivada, o clase hija.

Redefinición de métodos

- En la clase derivada podemos redefinir (override, o sobremontar) métodos, lo cual corresponde a re-implementar un método de la clase base en la clase derivada.
- Si aún deseamos acceder la método de la clase base, lo podemos hacer utilizando la palabra super como referencia al padre.
- Notar que también usamos esta palabra reservada para invocar constructores de la clase base.

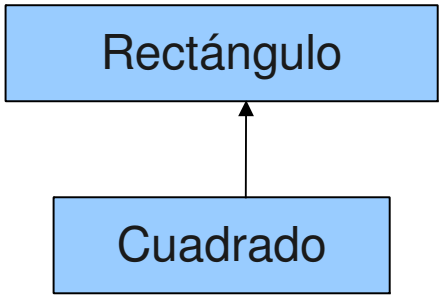
Ejemplo: Los gerentes también son empleados

- Supongamos que gerentes reciben bonos por su desempeño. Luego su salario será aquel en su calidad de empleado más los bonos que le correspondan.
- Ver ManagerTest.java
- Jerarquía de clases:



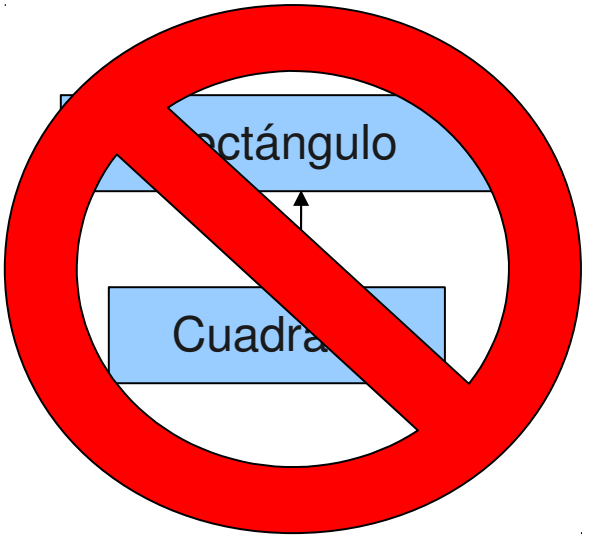
Principio de Sustitución

- **Recordar el principio de sustitución**: Según este principio, referencias a objetos de la clase base, pueden apuntar a objetos de una clase derivada sin crear problemas.
- Hay que tener cuidado con la relación es-un. El castellano permite decir que un cuadrado es un rectángulo de lados iguales; sin embargo, esto lleva a problemas cuando queremos aplicar el principio de sustitución.



?

Ejemplo: ¿Los cuadrados son rectángulos?



?

- Por ejemplo veamos una posible implementación aquí: Rectangle.java
- ¿Qué hay de la memoria ocupada si una aplicación requiere muchos cuadrados?
- ¿Qué pasa si recibidos una referencia a rectángulo y se nos ocurre invocar un cambio en uno de los lados?
- Lo podemos arreglar con redefinición de métodos, pero qué pasa con el uso natural que daríamos a rectángulos?

Polimorfismo

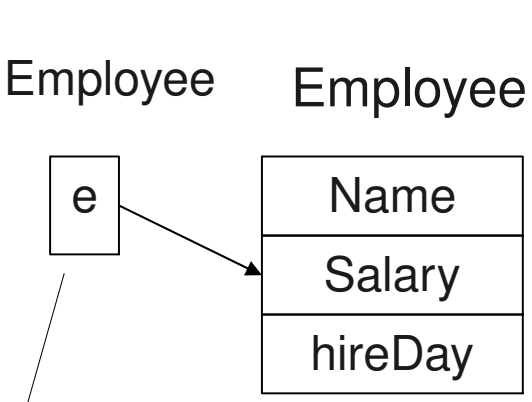
- Hay varias formas de polimorfismo:
  - Cuando invocamos el mismo nombre de método sobre instancias de distintas clases
  - cuando creamos múltiples constructores
  - cuando vía subtipo asignamos una instancia de una subclase a una referencia de la clase base.
- Lo último se explica porque cuando creamos una clase derivada, gracias a la relación es-un podemos utilizar instancias de la clase derivada donde se esperaba una instancia de la clase base. Lo vimos como principio de sustitución
- ¿Cómo podemos asignar un objeto que tiene más datos a uno que tiene menos?

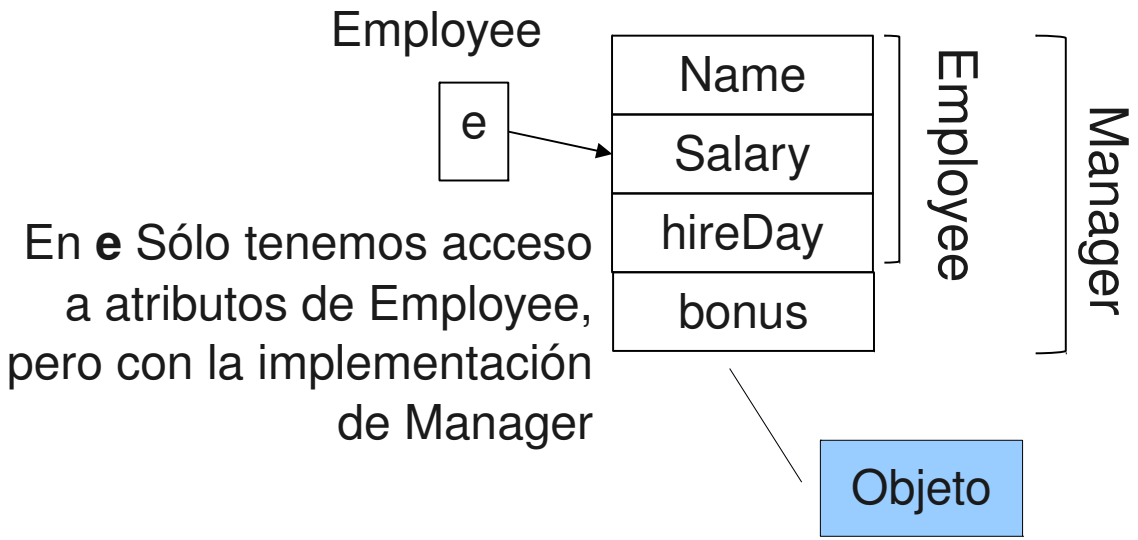
Polimorfismo: Ejemplo

- Sea:
  class Employee { ..... }
  class Manager extends Employee { .... }
- Employee e;
  e=new Employee(...); // instancia, **caso a**
  e=new Manager(..); // OK por Sustitución, **caso b**
- En el **caso a** usando **e** tenemos acceso a todo lo correspondiente a un Employee.
- En el **caso b** tenemos acceso a todo lo correspondiente a Employee, pero con la implementación de Manager.
- Al revés no es válido porque toda referencia a Manager debe disponer de todos los campos.

Polimorfismo: Ejemplo

- Sea:
  class Employee { ..... }
  class Manager extends Employee { .... }
- Employee e;
  e=new Employee(...); // instancia, **caso a**
  e=new Manager(..); // OK por Sustitución, **caso b**





Referencia Identificador o Nombre

Objeto

Ligado Dinámico

- Es importante entender qué método es usado al invocar a un nombre que se puede referir a instancias de clases derivadas.
- Al momento de la compilación el compilador intenta resolver el método que corresponde según su nombre y parámetros. Si la superclase y la clase base tienen definido el mismo método ¿Cuál se llama?.
- Si el método en la clase declarada para la referencia no es privado, static, o final, se invocará en forma dinámica.
- Esto es, se invocará el método definido según el objeto referenciado por el nombre y no según la declaración del nombre (que representa una referencia). A esto se le llama ligado dinámico.
- Por ello, si una clase derivada redefine el mismo método, éste será invocado para sus instancias.
- El ligado dinámico se resuelve a tiempo de ejecución, lo cual toma algo de tiempo.

Ligado Dinámico (cont.)

- Gracias el Ligado dinámico es posible hacer programas fácilmente extensibles.
- Creamos una clase derivada y redefinimos los comportamientos que deseamos.
- No se requiere recompilar las clases existentes. Esto es usado intensamente cuando utilizamos clases predefinidas en el lenguaje.
- Si deseamos impedir que una de nuestras clases se use como base, la declaramos como final.
  final class Manager extends Employee { ... }
- Si un método es final, ninguna subclase puede redefinirlo.
- El ligado dinámico es más lento que el estático.

Compilación v/s Ejecución

- El compilador verifica que los accesos y métodos invocados estén definidos en la clase declarada para el identificador o nombre del objeto usado, o alguna de sus superclases.
- En tiempo de ejecución, en código ejecutado depende de la declaración del método invocado. Si corresponde ligado dinámico, el código ejecutado será el del objeto apuntado por la referencia.
- Debemos distinguir entre la clase de la referencia o identificador y la clase del objeto asignado a la referencia.

Valores retornados por métodos redefinidos

- En la redefinición de un método, el nombre y los parámetros se deben conservar; no así el valor retornado.
- Cuando redefinimos un método en la clase derivada, la clase retornada puede diferir de aquella en el método de la clase base.
- Se debe cumplir que el objeto retornado por la clase derivada sea subtipo del de la clase base.

“Casteo”: Cambio de tipo forzado

- ¿Cómo podemos acceder a los métodos definido en una clase derivada con una referencia de la clase base?
- Se debe hacer un cambio de tipo forzado.
- Por ejemplo:

  Employee e= new Manager(..);

  - Con **e** no podemos acceder a los métodos presentes sólo en Manager.
  - Si queremos hacerlo, usamos:

  Manager m = (Manager) e;

  - Ahora con m sí podemos invocar los métodos de sólo en Manager.

“Casteo”: Cambio de tipo forzado (cont.)

- ¿Cómo sabemos que **e** es una referencia a una instancia de Manger?
- Lo podemos preguntar con el operador instance of.

```
if (e instanceof Manager) {
    m = (Manager) e;
.....
}
```

Clases abstractas

- Llevando la idea de herencia a un extremo, podemos pensar en buenas clases para representar un grupo de objetos que no tienen instancias propias.
  - Por ejemplo ElementoFisico como clase base para bloque y resorte; o Forma como clase base de Triangulo, Circulo, Cuadrado.
- ElementoFisico puede indicar todo el comportamiento válido para un elemento pero no se puede instanciar por si mismo. No tiene sentido instanciar una clase para la cual no se tiene todos los métodos implementados.
  Es decir no podemos hacer new Clase(), cuando Clase es abstracta.

Clases abstractas (cont.)

- En ente caso Forma debe declararse como clase abstracta por tener al menos un método declarado pero no implementado.

```
public abstract class ElementoFisico {
    ...
    public abstract String getDescripcion();
    ..
}
```

- ver PersonTest.java
- ver Gatos y perros

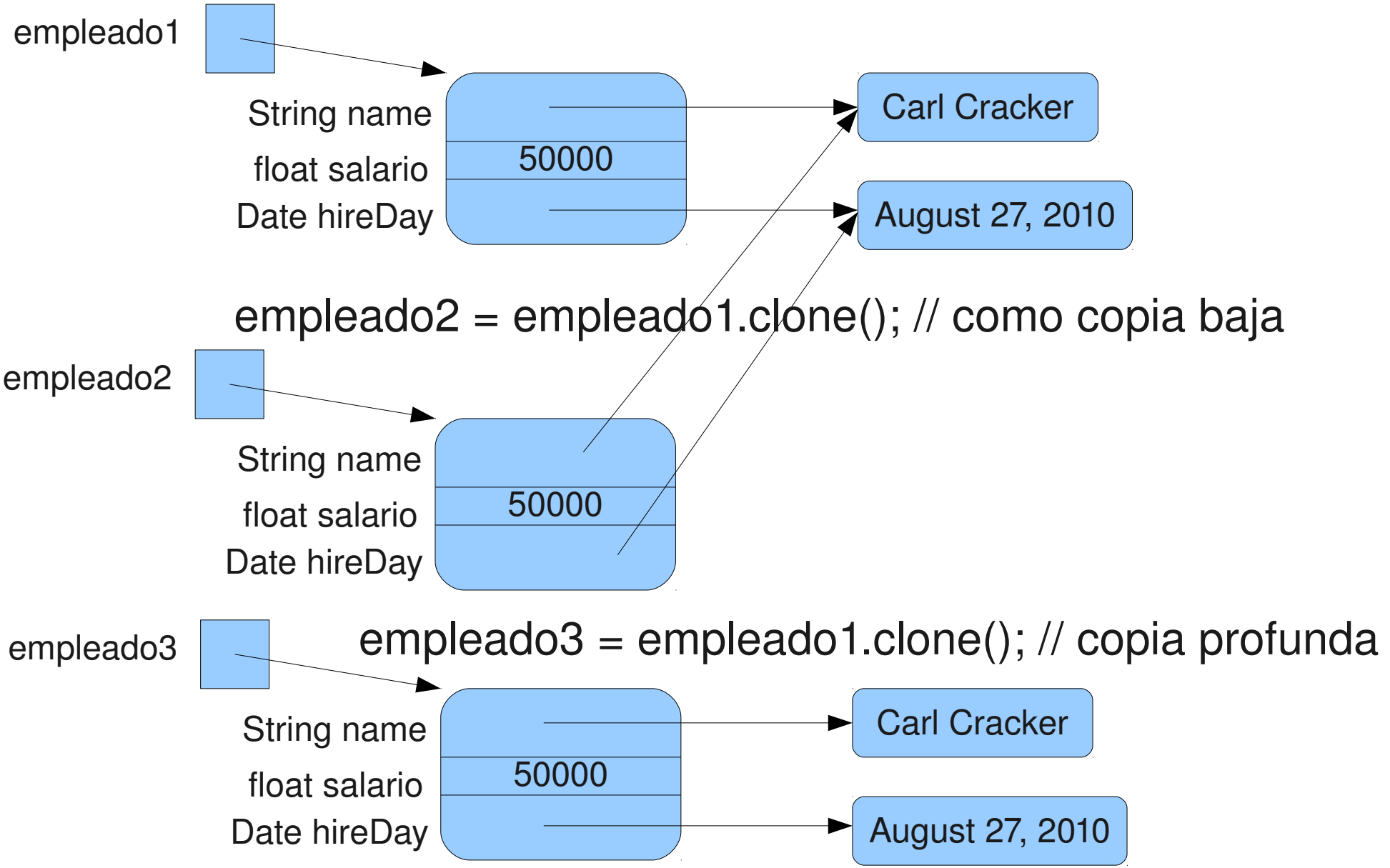
Clase Object: Nivel máximo de la jerarquía de clases

- Toda clase en Java hereda, en su jerarquía máxima, de la clase Object (ver en documentación).
- Ésta no requiere ser indicada en forma explícita.
- Esto permite que podamos agrupar en forma genérica elementos de cualquier clase, por ejemplo en un arreglo de Object.
- En esta clase hay métodos como equals() y toString() que en la mayoría de los casos conviene redefinir. ver documentación de clase Object. Ver: EqualsTest.java

Clase Object: método clone()

- 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, veremos interfaces más delante, 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



Implementación de clone (copia profunda)

- La implementación típica es como sigue:

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

Hasta aquí copia baja

Necesarios para copia profunda

Programación genérica/diseño de patrones

- Las facilidades que ofrece el diseño orientado a objetos y la programación orientada a objetos permiten ofrecer soluciones genéricas.
- La idea es poder crear código útil para varias situaciones similares.
- Por ejemplo podemos definir una clase con métodos como:

```
static int find (Object [ ] a , Object key)
{
    int i;
    for (i=0; i < a.length; i++)
       if (a[i].equals(key) return i;   // encontrado
    return -1;        // no exitoso
}
```

ArrayList: como Ejemplo de programación genérica

- Hay muchas estructuras de datos que no quisiéramos programar cada vez, ejemplo: stack, hash, lista, etc.
- El ArrayList permite crear arreglos de tamaño variable (ver ArrayList en documentación) .
- Lo malo es que el acceso no es con [ ].
- Ver esta clase en documentación

La clase Class

- La máquina virtual Java mantiene información sobre la estructura de cada clase. Ésta puede ser consultada en tiempo de ejecución.

```
Employee e = new Employee(...);
...
Class cl=e.getClass();
```

- La instancia de Class nos sirve para consultar datos sobre la clase, por ejemplo, su nombre.
  - System.out.println(e.getClass().getName()+" " +e.getName());
  - genera por ejemplo:

```
Employee Harry Hacker
```

La clase Class (cont.)

- Ver la clase Class. Nos permite obtener toda la información de una clase, su clase base, sus constructores, sus campos datos, métodos, etc.
- Por ejemplo ver ReflectionTest.java
- Esta funcionalidad normalmente es requerida por constructores de herramientas más que por desarrolladores de aplicaciones.