Herencia y Clases Abstractas

Reutilización de código y código genérico

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 reutilizando código (métodos y atributos).
- Podemos agregar métodos y atributos para adaptar la clase a la nueva categoría de objetos o situación (la clase nueva).
- Java también permite consultar por la estructura de una clase (cuáles son sus métodos y atributos). A esto se le llama reflexión (Sabrán los animales que son animales?, el hombre sí tiene conciencia sobre su existencia. En Java se puede consultar por la naturaleza de cada objeto … -fuera del alcance de este curso-)

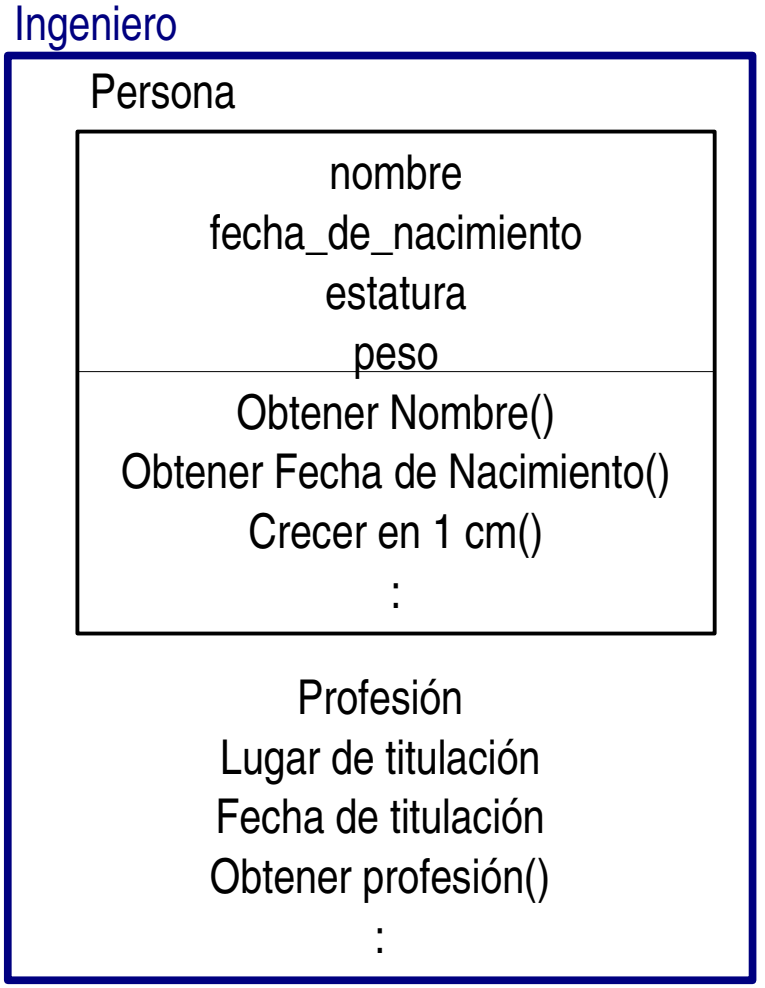
Introducción (cont.)

- 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 no 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 (son sinónimos aquí).
- A la nueva clase se le llama subclase, clase derivada, o clase hija.

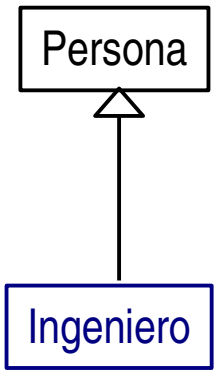
Ejemplo: Un Ingeniero es-una persona

Un ingeniero es una persona con propiedades específicas.

Un ingeniero es un caso particular de persona.
Hay otras personas como Médicos, abogados, carpinteros



Extendemos los métodos y/o atributos de Persona para llegar a un Ingeniero.



Representación UML de la relación de herencia

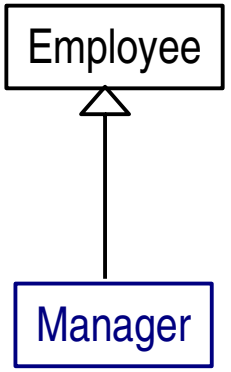
Aspectos sintácticos en Java

- Si un manager es un empleado y cumple el principio de sustitución, podemos definir manager extendiendo empleados:

```
class Manager extends Employee
{
  // aquí ponemos lo específico de un manager
  // que no está en todo empleado.
}
```

- Ver ejemplo: TestManager.java

Forma gráfica de representar la relación:

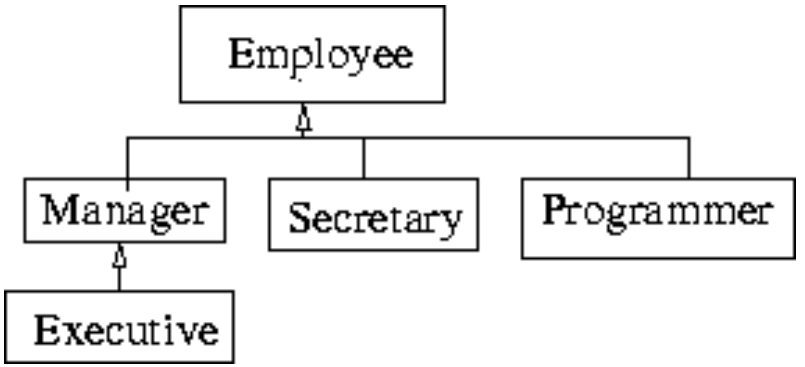


Redefinición de métodos

- 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 en la clase hija deseamos acceder al 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.

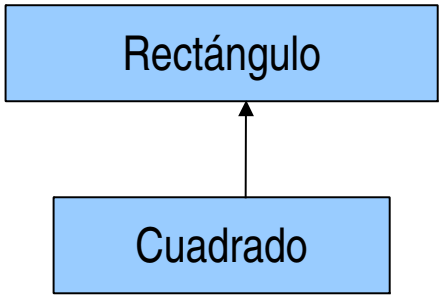
Ejemplo: Los gerentes también son empleados

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



Principio de sustitución revisitado (propiedad de subtipo)

- Según el principio de sustitución, 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 (rae: Cualidad de lo que tiene o puede tener distintas formas)

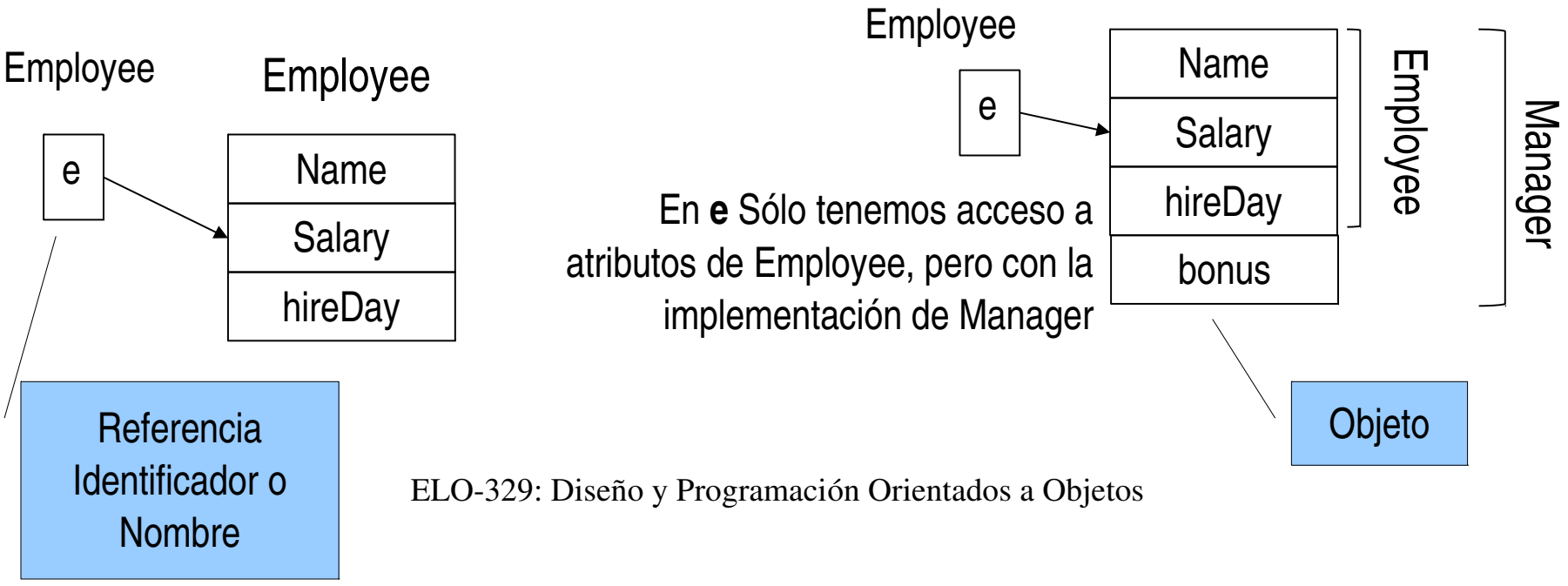
- Hay varias formas de polimorfismo:
  - Cuando vía subtipo asignamos una instancia de una subclase a una referencia de la clase base. Ej. Un ingeniero adopta el rol de persona.
  - Cuando invocamos el mismo nombre de método sobre instancias de distintas clases
  - Cuando creamos múltiples métodos de igual nombre
- La primera forma de polimorfismo listada corresponde a Subtipo o principio de sustitución.
- ¿Cómo podemos asignar un objeto que tiene más atributos a uno que tiene menos? No hay problema cuando ambos son referencias (punteros)..

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 (asignar un empleado a una referencia a Manager) 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**



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. Éstas no se pueden derivar.
  final class Manager extends Employee { ... }
- Si un método es final, ninguna subclase puede redefinirlo.
- El ligado dinámico es más lento que el estático (> tiempo de ejecución).

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, el 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 y la clase del objeto apuntado por 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.

Ej: Employee getColleague() {…} // en clase base

Manager getColleague() {…} // en clase hija.

"Casteo": Cambio de tipo forzado

- ¿Cómo podemos acceder a un método definido en una clase derivada con una referencia de la clase base?
- Se debe hacer un cambio de tipo forzado, cuando sabemos que el objeto es una instancia de la clase hija.
- 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, pero que su descripción es incompleta al depender de cada subclase.
  - Por ejemplo ElementoFisico como clase base para bloque y resorte; o Forma como clase base de Triangulo, Circulo, Cuadrado; o Compuerta como clase base para And, Ord, Not.
- 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 (ej. dibujese()).
  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

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

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 método como:

```
static int find (Object [ ] a , Object key)  //buscar en un arreglo
{
    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 ejemplo Cats and Dogs.

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.