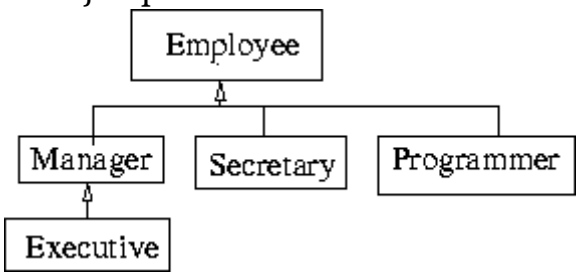
Herencia en Java

La idea de la herencia es permitir la creación de nuevas clases basadas en clases existentes.
Cuando heredamos de una clase existente, reusamos (o heredamos) métodos y campos, y agregamos nuevos campos y métodos para cumplir con la situación nueva.
Cada vez que encontremos la relación "es-un" entre dos clases, estamos ante la presencia de herencia. Atención con el lenguaje, podemos decir un cuadrado es un rectángulo de lados iguales; pero no es buena idea aplicar herencia cuando la nueva clase se crea restringiendo la existente. Si lo hacemos la solución no sólo ocupa más espacio de memoria sino también conduce a estructuras poco naturales o que conducen a resultados inesperados.

La clase ya existente es llamada superclass, o clase base, o clase padre.
La clase nueva es llamada subclase, clase derivada, o clase hija.

A través de herencia podemos agregar nuevos campos, y podemos agregar o sobre montar métodos (override). Sobre montar un método es redefinirlo en la case heredada. Para ello el nuevo método debe tener el mismo nombre, los mismos argumentos y el valor retornado debe ser igual o subclase del retornado por el método en la clase base.

Estudio de ejemplo: ManagerTest.java
Destacar uso de **super** para invocar al constructor de la clase base y para invocar a métodos sobre montados.
Jerarquía de Herencia en este ejemplo:



Polimorfismo

El polimorfismo se refiere a cuando una misma palabra se refiere a distintas cosas dependiendo de su uso en contexto. En Java se presenta de varias formas. Ya vimos una forma de polimorfismo al sobrecargar un método. Aquí usamos el mismo nombre de método pero al usar diferentes argumentos estamos invocando la ejecución de otro código. Otra forma de polimorfismo se logra a través del uso de subtipos.
En Java una variable o nombre usado para referirse a un objeto de una clase X puede usarse para referirse a cualquier objeto de cualquier subclase de la clase X. En este caso dependiendo de la ejecución del programa un nombre puede hacer referenica a objetos de distinta clase en la medida que ellos seas instancias de clases derivadas de la referenciada por ese nombre en su definición.

Por ejemplo: Si tenemos Employee e; y Manager es una clase heredada de Employee, entonces podemos hacer:
e= new Employee(...); o
e= new Manager(...);
Esto también aplica a arreglos de Employee, cuyas entradas podrían referirse a instancias de cualquier subclase de Employee.

El inverso no es válido. No se puede asignar una instancia de la superclass a un nombre de objetos de la subclase.
En el ejemplo equivale a que una referencia a Manager no puede apuntar a un Empleado. Esto arroja un error de compilación. Es natural pues un manager le popdemos pedir cosas (invocar métodos de Manager) que un emplado no podrá atender (no están definidos en Employee).

Ejemplo de apoyo: En la universidad hay Alumnos, los Electrónicos son subclase de alumnos. Si alguien necesita a un alumno, significa que espera de él cosas de tipo general que cualquier alumno puede atender. Si a la mano tenemos a un Electrónico, por ser alumno, también podrá cumplir con lo pedido. A quien lo pide no le interesa que sea un Electrónico, pues él pedirá servicios que cualquier alumno atendería - y también lo hará un Electrónico.

Para el caso de más arriba: a **e** sólo le podemos invocar métodos que estén en la clase Employee, que es la clase que él referencia. Cuando **e** referencie a un Manager y pidamos a **e** informarnos de su salario, estaremos invocando el método para el salario en Manager. Si **e** referencia a una instancia de Empleado, estaremos invocando el método para el salario en Empleado.

Vuelvo al ejemplo de apoyo, si al pedir un alumno nos llega un Electrónico, y le preguntamos ¿conoce usted la transformada de Laplace? seguro que contestará sí. Si se tratara de un alumno no electrónico, su respuesta podría ser no.

Esta es una forma de polimorfismo porque **e** aquí puede referirse a un Employee o a un Manager.

Ligado Dinámico
Es importante entender qué método es aplicado al invocar a un objeto que se puede referir a instancias de distinta clase.
Al momento de la compilación el compilador intenta resolver el método que corresponde según su nombre y parámetro. Si la superclass y la clase base tiene definido el mismo método ¿Cuál se llama?. Si el método en la clase declarada para la variable 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**. Por ello, si una clase derivada redefine el mismo método, éste será invocado.

Casting

¿Cómo podemos acceder a los métodos definido en una clase derivada pero no en la base? Se debe hacer un cambio de tipo forzado.

Por ejemplo:

```
Employee e= new Manager(..);
```

Usando **e** no podemos acceder a los métodos sólo presentes en Manager.

Si queremos hacerlo, usamos:

```
Manager m = (Manager) e;
```

Ahora sí podemos invocar todos los métodos de Manager sobre m. ¿Cómo sabemos que **e** aloja una instancia de Manager? Lo podemos preguntar con el operador **instanceof**.

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

Clases Abstractas (abstract classes)

Llevando la idea de herencia a un extremo podemos pensar una buena clase para representar un grupo de clases pero que no tienen instancias propias. Por ejemplo Forma como clase base de Triangulo, Circulo, Cuadrado. Forma puede indicar todo el comportamiento válido para una forma pero no puede instanciarse por si misma. En ente caso Forma debe declararse como clase abstracta. **Una clase debe declararse abstracta si tiene al menos un método declarado pero no implementado.** Esto ocurre cuando deseamos tener un método en todas las subclases, pero no podemos asignarle un código útil en la clase base.

```
public abstract class Forma {
  ...
  public abstract double area();  /* no implementado, se espera lo haga la clase derivada*/
  ..
}
```

ver PersonTest.java

Perros y gatos

Super Clase Object

Toda clase en Java hereda de la clase Object. Ésta no requiere ser indicada en forma explícita. Esto permite agrupar en forma genérica elementos de cualquier clase. En esta clase hay métodos como equals() y toString() que en la mayoría de los casos conviene sobre montar. ver documentación de clase Object.

Ver (alumnos) EqualsTest.java

Programación Genérica (Generic Programming)

Se le llama al hecho que podamos crear código válido para cualquier tipo de dato

específico. Por ejemplo en alguna clase podríamos incorporar:

```
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
}
```