cosas a tener en cuenta
resumen
definiciones y ejemplos

Herencia

Uno de los mecanismos más interesante para el diseño de clases que proveen los lenguajes basados en el paradigma de la Programación Orientada a Objetos es el de la herencia, ya que permite crear nuevas clases a partir de otras clases ya existentes.

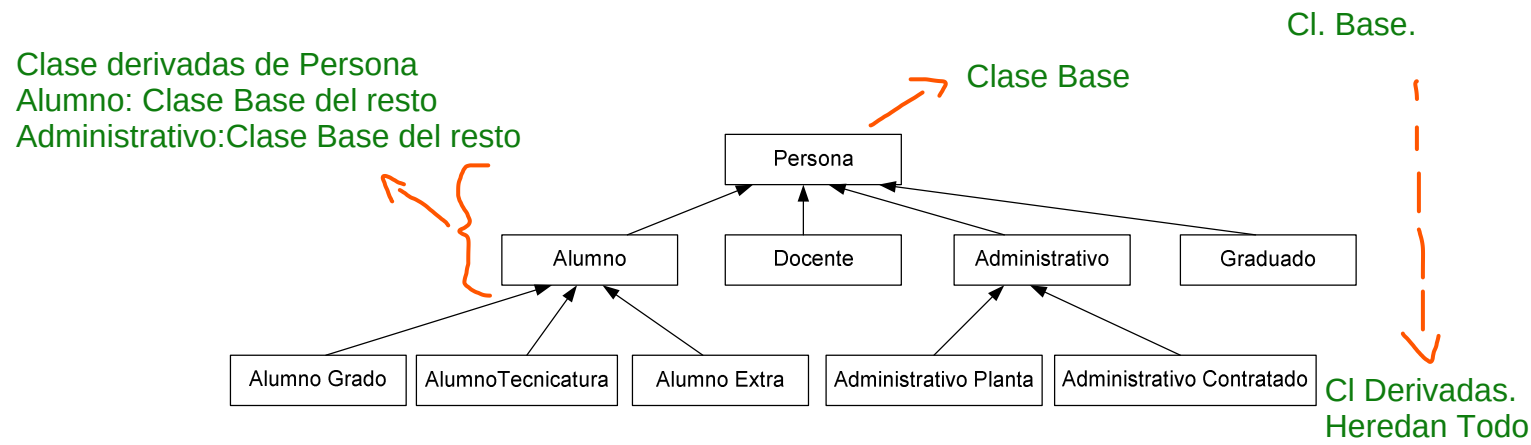
A las clases existentes, las que se usarán para el diseño de otras clases, se las llama **clases bases**. Las clases que se crearán a partir de las clases bases se denominan **clases derivadas**. Estas últimas heredarán todo lo que se haya definido en la clase base (propiedades y métodos), y en ellas se agregarán las propiedades y métodos faltantes. La única excepción es la establecida para los constructores y destructores de las clases bases, que no existirán como miembros en las clases derivadas.

En concreto, podemos pensar en el mecanismo de la herencia como si las propiedades y métodos de la clase base hubieran sido escritos en todas las clases que derivan de ella.

Una de las ventajas, y que aparece a simple vista, es que mediante el uso de la herencia se **reutiliza el código**, reduciendo el tiempo de trabajo en el diseño y la creación de clases; otra es la posibilidad de **propagar los cambios**, ya que todo aquello que se agregue o modifique en una clase base de manera inmediata pasará a formar parte de las clases que derivan de ella. Adicionalmente la herencia es utilizada en una técnica muy potente llamada polimorfismo, que parte del hecho de que un objeto de una clase derivada también es un objeto de la clase base, permitiendo realizar implementaciones generales capaces de ser aplicadas al conjunto de objetos pertenecientes a una misma familia de clases o jerarquía de herencia.

Las clases derivadas pueden, a su vez servir de clase base para otras clases derivadas, generando de esa manera una estructura jerárquica que se proyecta desde una clase base inicial padre, pasando por un conjunto de clases hijas, luego nietas, bisnietas, etc.

Veamos un ejemplo. Si tuvieramos que definir los objetos que interactúan en una universidad para representarlos en un sistema, inmediatamente pensaríamos en alumnos, graduados, docentes, administrativos, aulas, materias, carreras, entre otros. Luego, es probable que nos detengamos a considerar qué tienen en común esos objetos y que lleguemos a la conclusión que tanto los alumnos, como graduados, docentes y administrativos pertenecen a una categoría de orden superior que los contiene, ya que todos son personas. A su vez, puede que avanzando en nuestro análisis lleguemos a la conclusión que no existe un solo tipo de alumnos, ya que éstos pueden ser de carreras de grado, de tecnicaturas, y de cursos extracurriculares; del mismo modo dentro del personal administrativo podrían distinguirse los empleados de planta permanente y los contratados. Sin ánimo de hacer un análisis completo de las personas relacionadas con la Universidad, ya que no es nuestro objetivo, podríamos representar la familia de las personas descriptas con el siguiente diagrama:



El diagrama nos permite representar las jerarquías de relaciones que existen: Persona es la categoría que contiene a todas las personas. Independientemente de su rol (alumno, docente, administrativo) todos son personas; Alumno es una persona que representa a un conjunto que a su vez se diferencia por el tipo de estudio que cursa; y Administrativo a su vez se divide en planta y contrato. Docente y Graduado no necesitan diferenciaciones.

Pensando ahora en las definiciones que más arriba se hicieron del mecanismo de la Herencia, y en su aplicación al ejemplo podríamos decir que:

- Persona será la CLASE BASE para Alumno, Docente y Administrativo, que serán a su vez sus clase derivadas.
- Alumno será la CLASE BASE para Alumno Grado, Alumno Tecnicatura y Alumno Extra, que serán sus clase derivadas.
- Docente no tiene clases derivadas
- Administrativo será la CLASE BASE para Administrativo Planta y Administrativo Contratado, que serán sus clases derivadas.

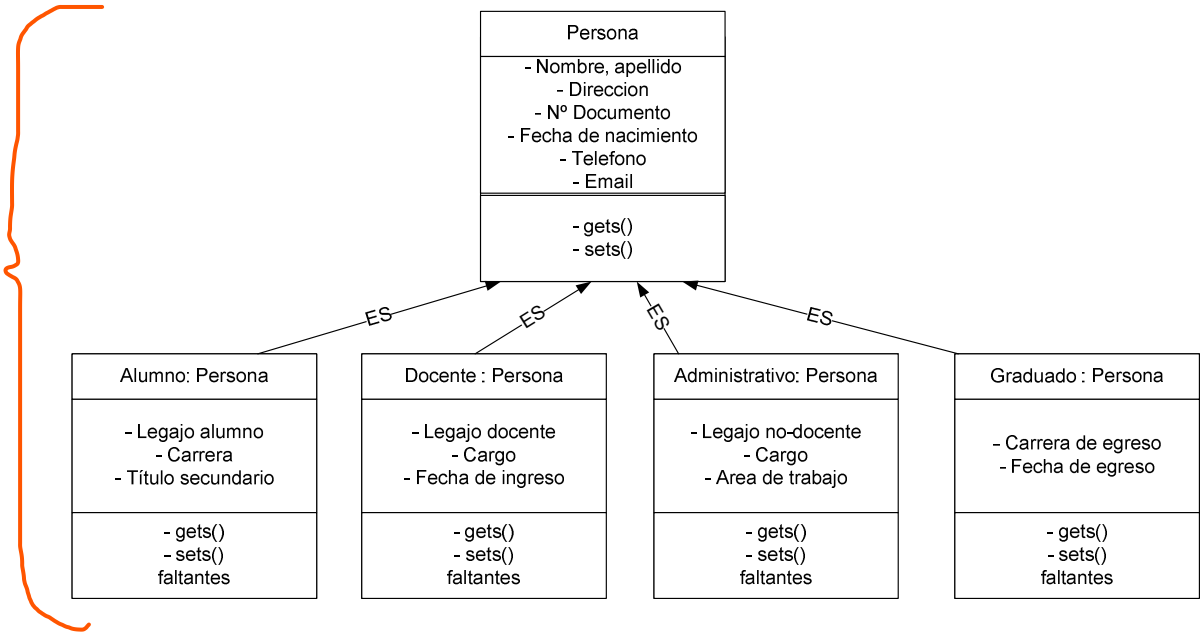
Ahora,¿en donde reside la potencia de la herencia?. Supongamos que para todas las personas necesitemos las siguientes propiedades:

Nombre
Apellido
Dirección
Nº de documento
Fecha de nacimiento
Nº de teléfono
Dirección de email

y como métodos los correspondientes gets y sets para cada propiedad, significaría que en cada una de las clases que representan distintos roles de personas deberíamos definir todas las propiedades y todos los métodos. Además puede suceder que en el proceso de refinamiento del diseño se agreguen o quiten propiedades, lo que obligaría a repetir el trabajo en cada una de las clases (son 6 si sólo analizamos las del último nivel). Al aplicar el mecanismo de la herencia sólo es necesario hacer la definición de las propiedades y el comportamiento en común en la clase base para que las clases derivadas las reciban y la puedan manipular como propias. Del mismo modo cualquier cambio que se haga en la clase base se propagará de inmediato hacia las clases derivadas.

Las clases derivadas sólo deberán agregar las propiedades y los métodos que le son propios y la diferencian del resto de las clases que, en este caso, representan a personas.

Una versión resumida del ejemplo anterior podría ser la siguiente:



La notación Alumno : Persona significa que Alumno deriva de Persona. Genéricamente sería Clase derivada : Clase Base.

Para el caso presentado, Alumno contaría con todas las propiedades y los gets() y sets() definidos en Persona, más las propiedades Legajo alumno, Carrera y Título secundario agregadas de manera particular dentro de la propia clase, y los gets() y sets() específicos para esas propiedades. Lo mismo para el resto de las clases.

Veamos como sería el código en C++

```
class Persona{
        protected:
                char ndoc[20];
                char nombre[20];
                char apellido[20];
                int dia, mes, anio;
                char domicilio[50];
                char telefono[20];
                char email[20];
        public:
                void Cargar();
                void Mostrar();
                const char * getNdoc(){return ndoc;}
                const char * getNombre(){return nombre;}
                const char * getApellido(){return apellido;}
```

Si fuera Private las clases derivadas no podrian heredar, en cambio con " PROTECTED"
Heredan sin romper el encapsulamiento.

```
int getDia(){return dia;}
int getMes(){return mes;}
int getAnio(){return anio;}
const char *getDomicilio(){return domicilio;}
const char * getTelefono(){return telefono;}
const char *getEmail(){return email;}
void setNdoc(char *n){strcpy(ndoc,n);}
void setNombre(char *n){strcpy(nombre,n);}
void setApellido(char *a){strcpy(apellido,a);}
void setDia(int d){dia=d;}
void setMes(int m){mes=m;}
void setAnio(int a){anio=a;}
void setDomicilio(cahr * d){strcpy(domicilio,d);}
void setTelefono(char *tel){strcpy(telefono,tel);}
void setEmail(char *e){strcpy(email,e);}
};
```

Para este caso en particular el único cambio que se ha realizado con respecto a una clase Persona que no se utilizara para derivar otras clases es el del especificador de acceso **private** por un nuevo especificador llamado **protected** que no habíamos utilizado anteriormente.

El especificador de acceso **protected** se utiliza debido a que, como se sabe desde el inicio del trabajo con POO, todo lo que se encuentra en una clase definido como private sólo será accesible dentro de la misma clase. Quiere decir que, aunque Alumno herede las propiedades de Persona, si en Persona estas propiedades son privadas no podrán ser accedidas desde Alumno. Con protected, en cambio, las clases derivadas sí pueden acceder a las propiedades que le son propias por herencia, manteniendo a la vez la protección que le da no ser accesibles desde fuera de la clases, como sería el caso que se definieran como public. **En resumen al usar protected los miembros siguen siendo privados en la clase base pero se permite el acceso a ellos por parte de las clases derivadas.**

```
class Alumno: public Persona
        {
         protected:// se define así debido a que esta clase se utilizará como clase base
                char nleg[15];
                char carrera[20];
                char titulo[30];
         public:
                void Cargar( );
                void Mostrar();
                const char * getLegajo(){return nleg;}
                const char * getCarrera(){return carrera;}
                const char * getTitulo(){return titulo;}
                void setLegajo (char *l){strcpy(nleg,l);}
                void setCarrera (char *c){strcpy(carrera,c);}
```

```
        void setTitulo (char *t){strcpy(titulo,t);}
};
```

La clase Alumno deriva en este caso de manera pública (public) de Persona. Esto significa que los miembros protegidos (protected) de Persona funcionarán como si estuviera escritos en la parte privada (o protegida si fuera el caso) de Alumno; los miembros públicos de Persona serán también públicos para Alumno.

Alumno podría haber derivado también de manera privada de Persona, de la siguiente manera:

Alumno : private Persona, o Alumno : Persona.

En este caso, todos los miembros de Persona funcionarán como si estuvieran en la parte privada o protegida de Alumno. Un tratamiento similar tendrían los miembros de Persona si Alumno derivara de manera protegida (Alumno: protected Persona)

Ahora podríamos declarar objetos de la clase Alumno, y contar con todas las propiedades de Persona más las de Alumno, y todos los métodos de Persona más los métodos de Alumno. Siguiendo esta idea, podemos afirmar que un objeto Alumno tiene dos métodos Mostrar(), y dos métodos Cargar(), esto es, tanto los definidos en Alumno como el anteriormente definido en Persona. ¿Será posible?

Efectivamente el objeto Alumno dispone de dos métodos Cargar() y dos Mostrar(), pero el que llama es el que pertenece a la clase del objeto, esto es, la clase derivada. Por ejemplo, en las líneas de código

```
----------
Alumno obj;
obj.Cargar();
 ----------
```

se declara un objeto Alumno, y el objeto llama al método Cargar() definido en la clase Alumno. Si se quisiera llamar al método Cargar() de Persona debería ponerse:

```
----------
Alumno obj;
obj.Persona::Cargar();
 ----------
```

Al tener dos metodos Cargar (Clase Base y Clase Derivada) Esta es la forma de llamar al Cargar de la Clase base desde la Derivada.

indicando que el método que quiere ejecutarse pertenece a la clase base Persona.

Sin embargo, probablemente lo que quisiéramos es que toda la funcionalidad de Cargar() esté definida dentro de la clase derivada sin necesidad de volver a escribir lo ya definido en Persona. El método sería entonces:

```
void Alumno::Cargar(){
        Persona::Cargar();// se llama al método existente que carga todos las
                          //propiedades de persona
        cout<<"INGRESE EL LEGAJO: ";
```

```
        cin>>nleg;
        cout<<"INGRESE LA CARRERA: ";
        cin>>carrera;
        cout<<"INGRESE EL TITULO: ";
        cin>>titulo;
        }
```

Para determinar si corresponde o no la aplicación del mecanismo de la herencia, puede procederse de la siguiente manera:

SI **CLASE DERIVADA ES CLASE BASE**, ENTONCES CORRESPONDE USAR HERENCIA

En nuestro caso, tanto Alumno, como Docente, Adminsitrativo y Graduado **SON** Personas.

Composición

Además de la herencia, podemos aplicar el mecanismo de la composición para la creación de clases. Un mecanismo no excluye al otro: en un mismo caso poder usar ambos mecanismos.

La composición consiste en utilizar un objeto de una clase como propiedad de otra clase.

Lo usamos por ej con la clase Fecha en los ejercicios. No es herencia pero al incluirla dentro de nuestra nueva clase podemos usar sus metodos y declarar un objeto tipo Fecha

Por ejemplo, en el caso de la clase Persona para representar una fecha se utilizan las propiedades dia, mes y anio; y para representar la dirección una cadena larga, cuando en realidad podríamos pensar en un objeto Fecha para la fecha de nacimiento, y un objeto Dirección para la dirección, porque ambas propiedades son en si mismas objetos compuestos por varios propiedades.

Para el caso de fecha, las propiedades son dia, mes y año; para la clase Dirección, las propiedades son:
Calle, número, CP, piso, número de departamento, etc. de acuerdo al detalle que necesitemos.

Además es altamente probable que necesitemos utilizar fechas y direcciones en otras clases, además de la que estamos analizando Persona, por lo que crear estas clases nos servirá también para el trabajo futuro.

Luego, la clase Persona quedaría de la siguiente manera:

```
class Persona{
        protected:
                char ndoc[20];
                char nombre[20];
                char apellido[20];
                Fecha fechaNac;
                Domicilio direccion;
                char telefono[20];
                char email[20];
        public:
                void Cargar();
                void Mostrar();
                const char * getNdoc(){return ndoc;}
                const char * getNombre(){return nombre;}
                const char * getApellido(){return apellido;}
                int getDia(){return fechaNac.getDia();}
                int getMes(){return fechaNac.getMes();}
                int getAnio(){return fechaNac.getAnio();}
                Domicilio getDireccion(){return direccion;}
                const char * getTelefono(){return telefono;}
                const char *getEmail(){return email;}
                void setNdoc(char *n){strcpy(ndoc,n);}
                void setNombre(char *n){strcpy(nombre,n);}
                void setApellido(char *a){strcpy(apellido,a);}
                void setDia(int d){fechaNac.setDia(d);}
                void setMes(int m){ fechaNac.setMes(m);}
                void setAnio(int a){ fechaNac.setAnio(a);}
                void setDireccion(Domicilio d){direccion=d);}
                void setTelefono(char *tel){strcpy(telefono,tel);}
                void setEmail(char *e){strcpy(email,e);}
                };
```

Composicion . Utilizamos una clase no declarada por herencia para usar sus metodos al incluirla en nuestra clase base

Notese que para manipular el objeto Fecha fechaNac se utilizaron los métodos de la clase Fecha, y que para direccion se recibe o devuelve un objeto de la clase Direccion. No debe confundirnos el hecho de que fechaNac y direccion sean propiedades de Persona: al ser objetos de otra clase deben ser manipulados con los métodos de sus propias clases.

Para determinar si corresponde utilizar la composición, la respuesta a :

CLASE TIENE OBJETO DE OTRA CLASE

debe ser verDadera.

En nuestro ejemplo:
Persona TIENE fecha de nacimiento
Persona TIENE dirección

En el Aula Virtual puede encontrarse un código de ejemplo completo con Herencia y Composición para la clase Persona, con el desarrollo de Fecha y Direccion.