Programación II

Herencia y Composición

Introducción a la Herencia

DEFINICIÓN:

En Programación Orientada a Objetos, la **herencia** permite que una clase (denominada ***derivada*** o ***subclase***) adquiera las ***propiedades*** y ***comportamientos*** de otra clase (llamada ***base*** o ***superclase***).

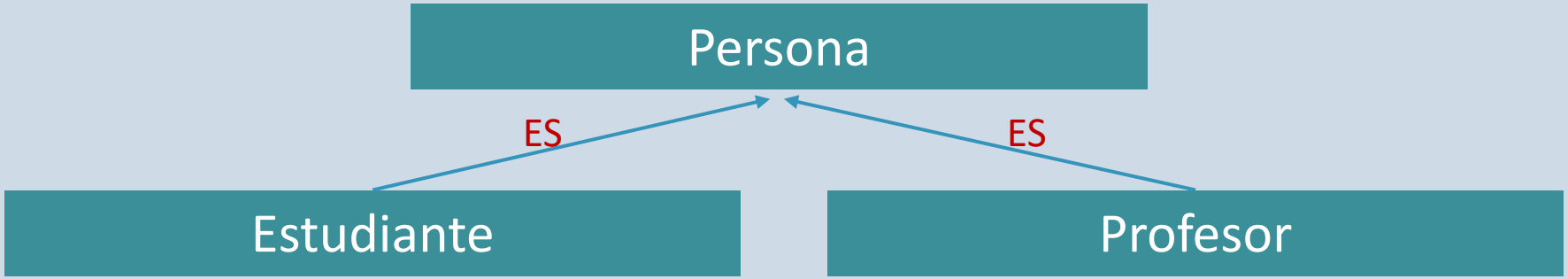
Ventajas:

- ✓**Reutilización de código**: Evita duplicación al compartir código común.
- ✓**Jerarquía lógica**: Organiza clases en relaciones de generalización-especialización.
- ✓**Mantenimiento**: Los cambios en la clase base se propagan automáticamente.
- ✓**Extensibilidad**: Facilita añadir nuevas funcionalidades.

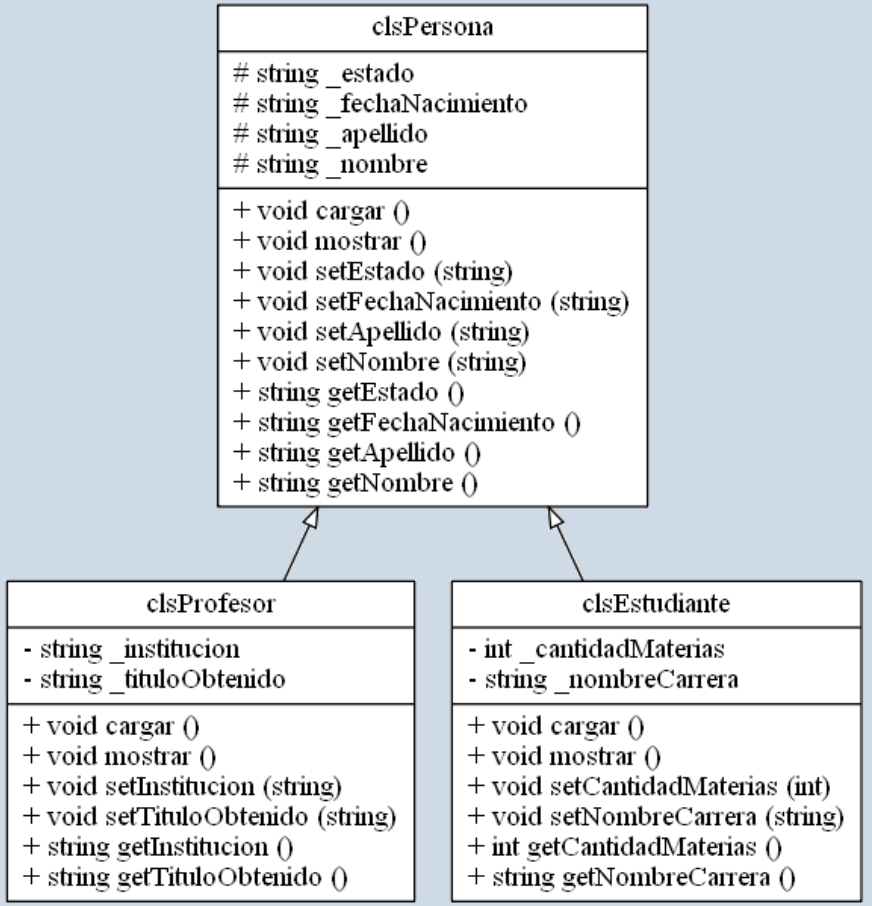
Herencia

- Una **subclase** hereda atributos y métodos de su **superclase**.
- Una clase derivada también puede convertirse en base de otras clases, formando jerarquías.
- La herencia centraliza código común en una única clase, reduciendo duplicación y simplificando el mantenimiento.

Herencia



Herencia


clsPersona
string _estado
string _fechaNacimiento
string _apellido
string _nombre
+ void cargar ()
+ void mostrar ()
+ void setEstado (string)
+ void setFechaNacimiento (string)
+ void setApellido (string)
+ void setNombre (string)
+ string getEstado ()
+ string getFechaNacimiento ()
+ string getApellido ()
+ string getNombre ()
clsProfesor
- string _institucion
- string _tituloObtenido
+ void cargar ()
+ void mostrar ()
+ void setInstitucion (string)
+ void setTituloObtenido (string)
+ string getInstitucion ()
+ string getTituloObtenido ()
clsEstudiante
- int _cantidadMaterias
- string _nombreCarrera
+ void cargar ()
+ void mostrar ()
+ void setCantidadMaterias (int)
+ void setNombreCarrera (string)
+ int getCantidadMaterias ()
+ string getNombreCarrera ()

Herencia

- En el ejemplo anterior, todas las personas del sistema deben tener: DNI, nombre, apellido, fecha de nacimiento y estado (activo o no), junto con sus getters y setters.
- Estas propiedades se definen una sola vez en la **clase base Persona**.
- Las **subclases** (como **Estudiante** o **Profesor**) **heredan** automáticamente estos atributos y métodos.
- Cada subclase solo agrega lo que la diferencia del resto, manteniendo claridad y facilitando el mantenimiento.

Herencia

- Por ejemplo, la clase **Estudiante** puede incluir atributos adicionales como “Carrera” o “Cantidad de materias inscriptas”.
- Del mismo modo, cada subclase puede tener características propias, además de las heredadas.
- Esto combina **reutilización** de código con **especialización** de cada tipo de objeto.

```
// Clase base
class clsPersona {
private:
    string dni, nombre, apellido;
    bool activo;
public:
    // Getters y setters para propiedades comunes
    void setNombre(string n) { nombre = n; }
    string getNombre() { return nombre; }
    // ... otros métodos comunes
};
```

```
// Clase derivada
class clsEstudiante : public clsPersona {
private:
    string carrera;
    int materiasInscriptas;
public:
    void setCarrera(string c) { carrera = c; }
    string getCarrera() { return carrera; }
    // Métodos específicos
};
```

```
// Uso en main()
int main() {
    clsEstudiante est;
    est.setNombre("María García"); // Heredado de clsPersona
    est.setCarrera(“Programación");  // Propio de clsEstudiante
    return 0;
}
```

Herencia (criterio de uso)

Para decidir si es adecuado aplicar el mecanismo de **herencia**, puede utilizarse el siguiente criterio:

Usar herencia cuando la relación entre clases sea del tipo ES

En nuestro ejemplo:

Un **Estudiante ES** una **Persona**

Herencia: ejemplos

Ejemplos válidos:

Un Estudiante **ES** una Persona ✓

Un Perro **ES** un Animal ✓

Un Auto **ES** un Vehiculo ✓

Ejemplos no válidos:

Un Empleado ES una Empresa ✗

(empleado pertenece a una empresa, pero no es un tipo de empresa)

Una Rueda ES un Auto ✗

(una rueda es parte del auto, no un auto en sí)

Composición

La **composición** es otro mecanismo de la Programación Orientada a Objetos que se utiliza para la construcción de clases.

No reemplaza ni excluye a la **herencia**, ya que en muchos casos ambos mecanismos pueden emplearse de manera complementaria.

La **composición** consiste en **incluir un objeto de una clase como propiedad dentro de otra clase**.

Composición

Ejemplo:

- En la clase Persona puede definirse una fecha de nacimiento.
- En lugar de guardar día, mes y año directamente, se crea una clase independiente Fecha.
- Persona incluirá un objeto de tipo Fecha para representar la fecha de nacimiento.

Ventajas:

- **Modularidad**: cada clase resuelve una única responsabilidad.
- **Reutilización**: la clase Fecha puede usarse en otros contextos además de Persona.

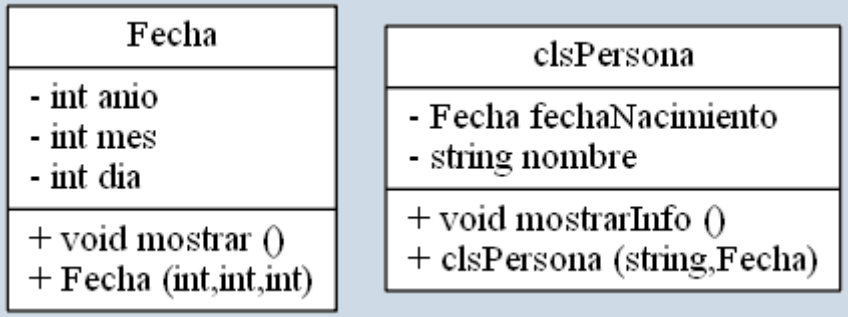
Composición

Estructura típica

```
class Componente {
    // Implementación del componente
};

class Contenedora {
private:
    Componente miComponente; // Composición
public:
    // Constructor que puede inicializar el componente
};
```

Composición



```cpp
class clsFecha{
private:
  int _dia, _mes, _anio;
public:
  // Constructor
  Fecha(int d, int m, int a);
  // Getters y setters
  // ... otros métodos
  void mostrar();
};
```

```cpp
// Clase base Persona
class clsPersona {
private:
    string _nombre;
    string _apellido;
    Fecha _fechaNacimiento;
    string _estado;

  public:
    // Getters
    string getNombre() { return _nombre; }
    string getApellido() { return _apellido; }
    Fecha getFechaNacimiento() { return _fechaNacimiento; }
    string getEstado() { return _estado; }
    // ... otros métodos
```

Composición (criterio de uso)

Para decidir si es adecuado aplicar el mecanismo de **composición**, puede utilizarse el siguiente criterio:

Usar composición cuando la relación entre clases sea del tipo TIENE

En nuestro ejemplo:

Persona TIENE una **Fecha** de nacimiento.