FACULTAD DE INFORMÁTICA

Más sobre herencia

TEMA

Programación orientada a objetos — Unidad 7

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Subclase **Alumno** con atributo de la clase **Persona**

El archivo **alumno.h**

```
#ifndef alumno_h
#define alumno_h

#include <iostream>
#include <string>
using namespace std;
#include "persona.h" // Inclusión de la clase Persona

// Todos los métodos implementados fuera
class Alumno : public Persona {
public:
  // Constructor (predeterminado):
  Alumno(Persona = Persona(), string = "", int = 0,
         string = "", string = "", int  = 1);
  // Profesor, NIF, edad, nombre, apellidos y curso
  Alumno(const Alumno&); // Constructor de copia
  Alumno& operator=(const Alumno&); // Copia
  ~Alumno(); // Destructor
```

(*continúa*)

FdI UCM

Subclase **Alumno** con atributo de la clase **Persona**

```
  void curso(int);
  void profesor(Persona);
  int curso() const;
  Persona profesor() const;
  void leer(); // No pide los datos del profesor
  void mostrar() const;
private:
  int _curso;
  Persona _profesor; // Atributo de clase Persona
};

#endif
```

Subclase **Alumno** con atributo de la clase **Persona**

El archivo **alumno.cpp**

```
#include "alumno.h"

Alumno::Alumno(Persona profesor, string nif, int edad,
  string nombre, string apellidos, int curso)
 : _curso(curso), _profesor(profesor),
 Persona(nif, edad, nombre, apellidos) { }

Alumno::Alumno(const Alumno& otro) : Persona(otro) {
  _curso = otro._curso;
  _profesor = otro._profesor;
}

Alumno& Alumno::operator=(const Alumno& otro) {
  Persona::operator=(otro);
  _curso = otro._curso;
  _profesor = otro._profesor;
  return *this;
}
```

(*continúa*)

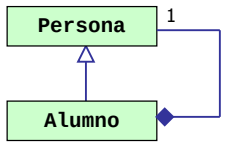
Subclase **Alumno** con atributo de la clase **Persona**

```
Alumno::~Alumno() { }

void Alumno::curso(int num) { _curso = num; }

void Alumno::profesor(Persona profe)
{ _profesor = profe; }

int Alumno::curso() const { return _curso; }

Persona Alumno::profesor() const { return _profesor; }

void Alumno::mostrar() const {
  Persona::mostrar();
  cout << "Curso: " << _curso << endl;
  cout << "Profesor:" << endl;
  _profesor.mostrar();
}
```

(*continúa*)

Subclase **Alumno** con atributo de la clase **Persona**

```
void Alumno::leer() {
  Persona::leer();
  cout << "Curso: ";
  cin >> _curso;
}
```



La clase **Alumno** es subclase de **Persona** y también contiene una **Persona** (el profesor)

Subclase **Alumno** con atributo de la clase **Persona**

Prueba de la clase **Alumno**

```
#include "persona.h"
#include "alumno.h"

int main()
{
  Persona profe("445566F", 33, "LUIS", "HERNANDEZ YAÑEZ");
  Alumno alumno1(profe);
  alumno1.leer();
  alumno1.mostrar();

  return 0;
}
```

Subclase **Alumno** con atributo de la clase **Persona**

Creación de objetos que contienen objetos

Debemos recordar que cuando en una clase (**Alumno**) se disponen atributos de otras clases (**_profesor**), cuando se crea un objeto de aquella (**Alumno**), antes de ejecutar automáticamente el constructor de la clase (y, antes, los de sus superclases) se crea el objeto atributo, lo que provoca la ejecución automática de su constructor.

En este ejemplo, cuando se crea el objeto de clase **Alumno**:

```
Alumno alumno1(profe);
```

se crea primero el atributo **_profesor** del objeto **alumno1**, ejecutándose sobre él el constructor de la clase **Persona**, la clase a la que pertenece.

Una vez hecho, se seguirá con la cadena de ejecuciones de los constructores adecuados para el objeto **alumno1** (primero, y de nuevo, el de la superclase **Persona**, pero sobre el objeto **alumno1**, y luego el de la propia clase **Alumno**).

Subclase **Alumno** con atributo de la clase **Persona**

También debemos tener presente que cuando se ejecuta el constructor de copia para crear por copia un objeto de la clase **Alumno**, al igual que cuando se copia un objeto **Alumno** en otro (operador de asignación), se ejecuta el código que copia el atributo **_profesor**:

```
_profesor = otro._profesor;
```

y eso significa que se ejecuta el operador de asignación de la clase **Persona**, clase a la que pertenece ese atributo **_profesor**.

Es muy importante tener muy claro qué es lo que se ejecuta en cada momento cuando se trabaja con los objetos, más aún teniendo en cuenta que se ejecutan de forma automática determinados métodos, como los constructores y los destructores.

Relaciones entre clases

En unidades anteriores vimos cómo se establecen las relaciones de clientelismo entre las clases (*una clase hace uso de objetos de otra*).

Hemos visto que también se puede establecer una relación de herencia entre las clases.

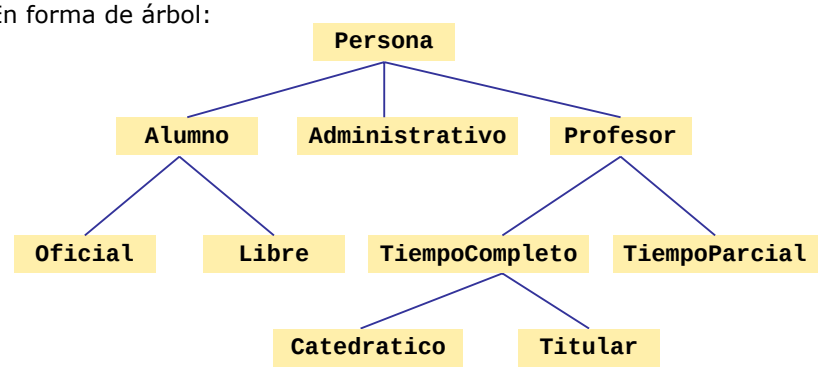
- ✓ Clientelismo: la clase A es cliente de la clase B si en la implementación de la clase A se hace uso de objetos de la clase B.
- ✓ Herencia: la clase A desciende (se deriva) de la clase B (mediante derivación directa -A es subclase de B- o mediante una cadena de derivación -A es subclase de una subclase de B, por ejemplo-).

Las relaciones entre clases establecen dependencias entre ellas que se han de contemplar en la reutilización: para utilizar una clase C se necesita no sólo la implementación de dicha clase, sino también la de todas aquellas otras de las que dependa (por herencia o por clientelismo).

Jerarquía(s) de clases

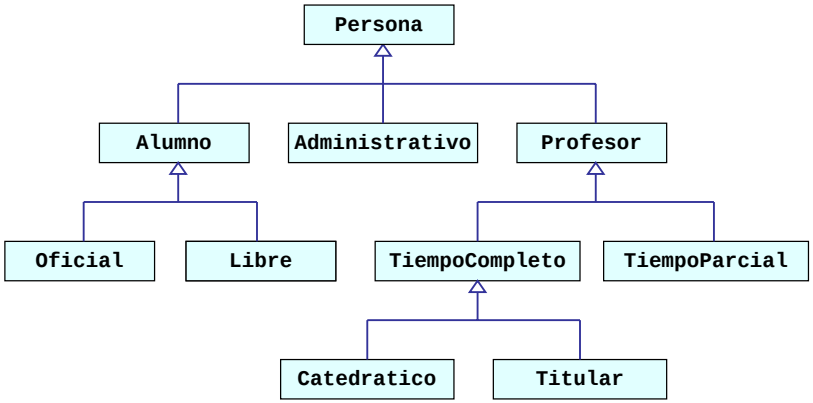
A medida que se establecen relaciones de herencia entre las clases, implícitamente se va construyendo la jerarquía de clases del sistema.

En forma de árbol:



Jerarquía(s) de clases

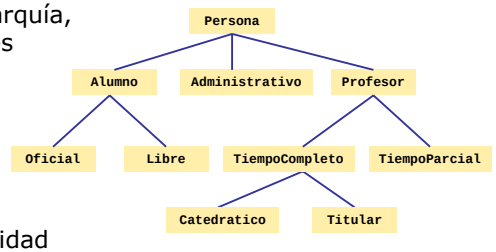
Utilizando diagramas de UML:



Jerarquía(s) de clases

Cuanto más arriba en la jerarquía, más abstractas son las clases (menor nivel de detalle).

A medida que se desciende por la jerarquía aumenta el nivel de detalle (disminuye la abstracción).



Así, una **Persona** es una entidad mucho más abstracta y general que un **Titular**.

- ✓ Cada clase de la jerarquía debe implementar todas las características que son comunes a todas sus subclases.
- ✓ Cada subclase debe contemplar únicamente las peculiaridades que la distinguen de su superclase.

En C++ habrá múltiples jerarquías (no hay una clase **Objeto** global).

Clases abstractas

En ocasiones tendremos definidas clases en la jerarquía que simplemente recogen las características comunes de otra serie de clases (sus descendientes), pero que no se van a (o no se deben) utilizar para crear ejemplares.

Clase abstracta:

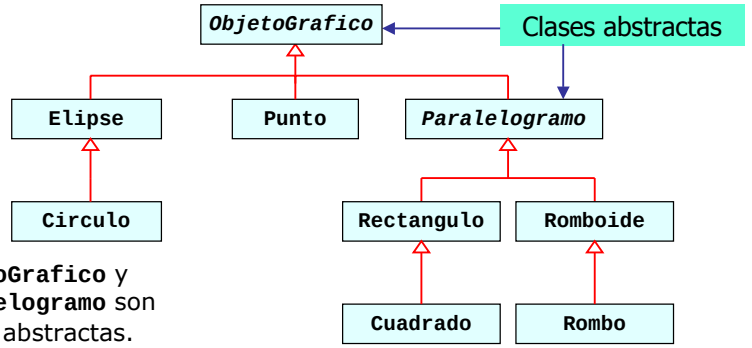
- ✓ Modela el comportamiento común de sus clases derivadas.
- ✓ Implementa métodos que son comunes a todas sus subclases.
- ✓ Establece métodos que necesariamente han de ser implementados por sus subclases (las clases derivadas).

En el sistema no van a crearse ejemplares de la clase abstracta porque no serían objetos con existencia propia en el mundo real.

Los objetos que se crearán serán ejemplares de las subclases (aquellas que no sean también abstractas).

La clase abstracta puede definir atributos comunes a sus subclases.

Clases abstractas



ObjetoGrafico y **Paralelogramo** son clases abstractas.

En el programa de dibujo sólo se van a crear objetos gráficos concretos: puntos, elipses, círculos, cuadrados, rectángulos, romboides o rombos.

Más adelante veremos cómo definir clases abstractas.

Ejecución de los pasos de mensajes con herencia

Ahora que hemos introducido el mecanismo de herencia debemos ver cómo afecta al mecanismo de paso de mensajes.

Cuando a un **objeto** se le pasa un **mensaje**:

1. Se busca en la clase del **objeto** un método definido con ese nombre **mensaje**. Si se encuentra y el número y tipo de los argumentos proporcionados en el paso de mensaje concuerdan, se ejecuta ese método y termina el proceso.
2. Si en la clase del **objeto** no hay ningún método definido con ese nombre **mensaje**, o si el número y tipo de los argumentos no concuerda, se busca en la superclase inmediata.
   Si se encuentra, se ejecuta y termina el proceso; si no, se busca en la siguiente superclase (subiendo por la jerarquía).
3. Si en el proceso de búsqueda del método se llega hasta la clase superior de la jerarquía y tampoco se encuentra allí, entonces *el **objeto** no entiende el **mensaje***.

Ejecución de los pasos de mensajes con herencia

```
Alumno unAlumno;
...
unAlumno.mostrar();
```

El método **mostrar()** está definido en la clase **Alumno**, por lo que se ejecuta dicho método (el mensaje **mostrar()** se vincula con el método **mostrar()** de la clase **Alumno**).

```
Alumno unAlumno;
...
unAlumno.felizCumple();
```

El método **felizCumple()** NO está definido en la clase **Alumno**, por lo que se busca en la superclase (**Persona**); allí sí se encuentra, ejecutándose ese método (el mensaje **felizCumple()** se vincula con el método **felizCumple()** de la clase **Persona**).

Ejecución de los pasos de mensajes con herencia

```
Alumno unAlumno;
...
unAlumno.pasarCurso();    // Error
```

No hay ningún método **pasarCurso()** definido en la clase **Alumno** ni en su superclase **Persona**, por lo que se concluye que el objeto **unAlumno** no entiende el mensaje **pasarCurso()** (el mensaje **pasarCurso()** no se puede vincular con ningún método).

En este caso se generará un error de compilación, ya que el intento de vinculación entre el mensaje y algún método se puede realizar en tiempo de compilación (*vinculación estática*).

En otras ocasiones, como veremos más adelante, la vinculación entre el mensaje y algún método sólo se puede realizar en tiempo de ejecución (*vinculación dinámica*); en esos casos, si no se puede realizar la vinculación se produce un error de ejecución.

Problemas con la vinculación de mensajes

```
int main()
{
  Persona p("445566F", 33, "LUIS", "HERNANDEZ YANEZ");
  Alumno al(p, "223344G", 21, "ROSA", "RATO PALOMO", 2);
  al.mostrar();
  cout << endl;
  al.felizCumple();

  return 0;
}
```

```
223344G
ROSA RATO PALOMO
Edad: 21
curso: 2
Profesor:
445566F
LUIS HERNANDEZ YANEZ
Edad: 33

Registro actualizado:
223344G
ROSA RATO PALOMO
Edad: 22
```

>>> prueba.cpp

Problemas con la vinculación de mensajes

En el programa todo va bien hasta que se llega al paso de mensaje **al.felizCumple()**.

El método **felizCumple()** no está definido en la clase **Alumno**, pero sí en la superclase **Persona**:

```
void Persona::felizCumple() {
  _edad++;
  cout << "Registro actualizado:" << endl;
  mostrar();
}
```

Se vincula el mensaje **felizCumple()** con el método **felizCumple()** de la clase **Persona**.

El método básicamente incrementa el atributo **_edad** y muestra la información del objeto.

Problemas con la vinculación de mensajes

Como se ve, la edad se incrementa correctamente.

Sin embargo, cuando se muestra la información con el paso de mensaje

mostrar();

se vincula incorrectamente el mensaje **mostrar()** con el método **mostrar()** de la clase **Persona**, en lugar de con el de la clase **Alumno**.

```
Registro actualizado:
223344G
ROSA RATO PALOMO
Edad: 22
```

_edad incrementada

Consecuentemente, se muestra sólo información sobre **_nif**, nombre completo y **_edad**; nada sobre **_curso** o **_profesor**.

Problemas con la vinculación de mensajes

Cuando se ejecuta el paso de mensaje, al encontrarse dentro del método **felizCumple()**, que es un método de la clase **Persona**, se asume de forma implícita que el objeto que ha recibido el mensaje **felizCumple()** es un ejemplar de esa clase **Persona**, por lo que **mostrar()** se vincula con ese método **mostrar()**.

Sin embargo, el objeto que recibió el mensaje **felizCumple()** es un ejemplar de la subclase **Alumno** y debería ejecutarse el método **mostrar()** de esa subclase.

Problema:
Se *olvida* la clase del objeto que recibe el mensaje original (**felizCumple()** en este caso) que se ha vinculado con el método.

Solución:
Utilización de funciones miembro virtuales que aseguren que se *sigue la pista* en todo momento a la identidad del objeto receptor (lo veremos más adelante).

Compatibilidad entre objetos de distintas clases

Todos los alumnos son personas,
pero no todas las personas son alumnos

A un identificador de la clase **Persona** (superclase) se le puede asignar un objeto de la clase **Alumno** (subclase), pero a un identificador de la clase **Alumno** (subclase) **no** se le puede asignar un objeto de la clase **Persona** (superclase)

```
Persona unaPersona;
Alumno unAlumno;
...
unaPersona = unAlumno; // Correcto:
                       // todos los alumnos son personas
unAlumno = unaPersona; // INCORRECTO:
                       // no todas las personas son alumnos
```

Compatibilidad entre objetos de distintas clases

Regla general de compatibilidad entre objetos:

> A un identificador de una clase sólo se le pueden asignar objetos de esa clase o de cualquiera de sus subclases

Esta compatibilidad resulta muy útil cuando se quieren manejar listas de objetos: basta declarar un array de objetos de la superclase y podremos asignar a las distintas posiciones del array objetos de esa clase y de cualquiera de las subclases.

Más problemas con la vinculación de mensajes

```
int main()
{
  Persona p("445566F", 33, "LUIS", "HERNANDEZ YANEZ");
  Alumno al(p, "223344G", 21, "ROSA", "RATO PALOMO", 2);
  al.mostrar();
  cout << endl;
  p = al;
  // ahora p es igual
  // a al (el alumno)
  p.mostrar();

  return 0;
}
```

```
223344G
ROSA RATO PALOMO
Edad: 21
curso: 2
Profesor:
445566F
LUIS HERNANDEZ YANEZ
Edad: 33

223344G
ROSA RATO PALOMO
Edad: 21
```


Más problemas con la vinculación de mensajes

En el programa todo va bien hasta que se llega al paso de mensaje **`p.mostrar()`**.

Justo en la instrucción anterior hemos asignado un objeto **`Alumno`** a **`p`**. La regla de compatibilidad lo permite, ya que **`p`** está declarado como de clase **`Persona`** (superclase de **`Alumno`**).

Sin embargo, al ejecutarse no aparece en la pantalla toda la información sobre el alumno:

```
223344G
ROSA RATO PALOMO
Edad: 21
```

¿Información de curso y profesor?

Más problemas con la vinculación de mensajes

En este caso, aparte de los problemas de tamaño, de los que hablaremos con más detenimiento al final de la unidad (**`p`** está preparado para guardar información sólo de **`Persona`**), tenemos nuevamente problemas de vinculación.

¿Qué método **`mostrar()`** hay que ejecutar? El método está definido tanto en la superclase como en la subclase. A **`p`** le acabamos de asignar un **`Alumno`**, por lo que esperaríamos que se ejecutara el método de la subclase **`Alumno`**. Sin embargo, el compilador lo único que mira es la declaración del identificador que recibe el mensaje. Como **`p`** es de clase **`Persona`**, no lo piensa dos veces y ejecuta el método de la superclase.

Esto es lo que hace por defecto, si no le decimos que haga otra cosa. Lleva a cabo una vinculación estática entre el mensaje y el método a ejecutar. Para que vincule dinámicamente (averiguando primero cuál es realmente la clase del objeto receptor), debemos nuevamente utilizar métodos virtuales (lo veremos más adelante).

Una lista de **Personas** y **Alumnos**

El archivo **listaper.h**

```
#ifndef listaper_h
#define listaper_h

#include "persona.h"

class Lista {
public:
  Lista();
  Lista(const Lista&);
  Lista& operator=(const Lista&);
  ~Lista();

  bool llena() const;
  bool vacia() const;
```

(*continúa*)

Una lista de **Personas** y **Alumnos**

```
  bool insertar(Persona);
    // Devuelve true si se inserta; false si lista llena
  bool recuperar(int, Persona&) const;
    // true si tiene éxito; false si posición no válida
  void mostrar() const;
private:
  enum { MAX = 100 };
  Persona _array[MAX];
  int _cont;
}; // Fin de la definición de la clase Lista

#endif
```


Una lista de **Personas** y **Alumnos**

El archivo **listaper.cpp**

```
#include <iostream>
#include <string>
using namespace std;
#include "listaper.h"

Lista::Lista() : _cont(0) {}

Lista::Lista(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i]; }

Lista& Lista::operator=(const Lista& otra) {
  _cont = otra._cont;
  for(int i = 0; i < _cont; i++)
    _array[i] = otra._array[i];
  return *this; }
```

(*continúa*)

Una lista de **Personas** y **Alumnos**

```
Lista::~Lista() {}

bool Lista::llena() const { return _cont == MAX; }

bool Lista::vacia() const { return _cont == 0; }

bool Lista::insertar(Persona p) {
  if(_cont == MAX) return false;
  _array[_cont] = p;
  _cont++;
  return true;
}

bool Lista::recuperar(int pos, Persona& p) const {
  if(pos < 1 || pos > _cont) return false;
  p = _array[pos-1];
  return true;
}
```

(*continúa*)

Una lista de **Persona**s y **Alumno**s

```
void Lista::mostrar() const {
  cout << "Elementos de la lista:" << endl;
  for(int i = 0; i < _cont; i++)
    _array[i].mostrar();
}
```

¿Qué ha cambiado respecto de la implementación de la clase **Lista** de la Unidad 5?

NADA. El hecho de que se puedan guardar tanto **Persona**s como **Alumno**s, que son objetos de una subclase está garantizado por la regla de compatibilidad de objetos y no es necesario cambiar nada. Si el array está definido sobre la clase **Persona**, automáticamente se pueden guardar también objetos de cualquiera de sus subclases.

Pero eso no quiere decir que no vayamos a encontrar problemas...

>>> listaper.cpp

Una lista de **Persona**s y **Alumno**s

```
int main()
{
  Persona persona;
  Alumno alumno;
  Lista miLista;
  int op;

  do {
    cout << "1 - Insertar persona" << endl;
    cout << "2 - Insertar alumno" << endl;
    cout << "3 - Listar" << endl;
    cout << "0 - Salir" << endl;
    cout << "Opción: ";
    cin >> op; cout << endl;
    switch(op) {
      case 1:
        if(miLista.llena())
          cout << "La lista ya esta llena" << endl;
```

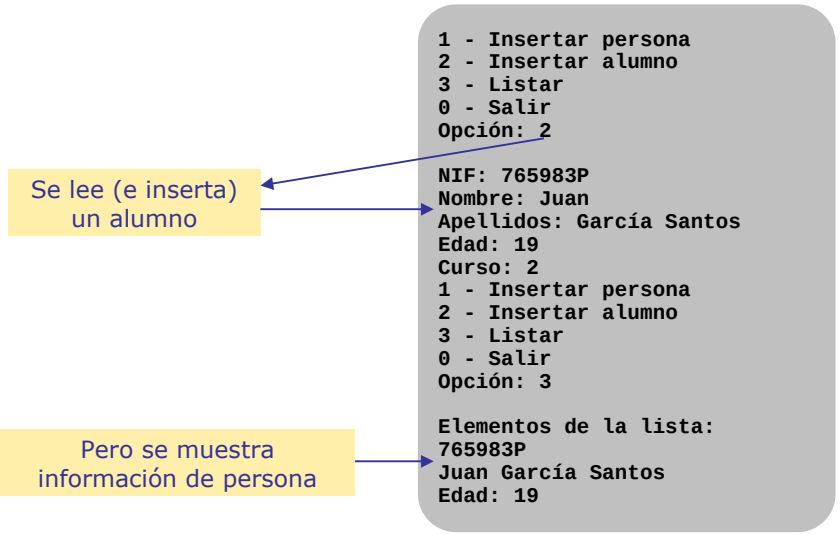
(*continúa*)

Una lista de **Persona**s y **Alumno**s

```
        else {
          persona.leer();
          miLista.insertar(persona);
        }
        break;
      case 2:
        if(miLista.llena())
          cout << "La lista ya esta llena" << endl;
        else {
          alumno.leer();
          miLista.insertar(alumno);
        }
        break;
      case 3:
        miLista.mostrar();
        break;
      }
    } while(op != 0);

  return 0;
}
```


>>> prueba2.cpp

Una lista de **Persona**s y **Alumno**s



Una lista de **Personas** y **Alumnos**

La regla de compatibilidad de objetos permite insertar **Alumnos** en la lista (declarada para albergar **Personas**).

El parámetro **Persona p** del método **insertar()** admite el objeto **alumno** de clase **Alumno** como argumento.

Se asigna sin problemas el argumento **alumno** a la siguiente posición libre del array **_array** (array de **Personas**).

Pero cuando se ejecuta la instrucción

```
miLista.mostrar;
```

y en el método **mostrar()** de la clase **Lista** se ejecuta

```
_array[i].mostrar();
```

cuando en la posición **i** se guardó un **Alumno**, no se ejecuta el método **mostrar()** de **Alumno**, sino el de **Persona**.

¿Cuál es ahora el problema?

Una lista de **Personas** y **Alumnos**

El problema es que el **_array** sólo tiene espacio físico para objetos de la clase **Persona**, por lo que cuando se inserta un objeto de clase **Alumno** se pierde la parte del objeto que no se corresponde con su *estatus* de **Persona**.

Es decir, en las celdas del array caben objetos de clase **Persona**.

Los objetos de clase **Alumno** tienen lo mismo que los de clase **Persona** y algo más (**_curso** y **_profesor**).

En las celdas del array cabe la *parte de **Persona*** que hay en los **Alumnos** y nada más.

La regla de compatibilidad de objetos permite asignar **Alumnos** a **Personas**, pero eso no quita que no se pueda perder información en el proceso de asignación.

La solución a este problema de pérdida de información se encuentra en el uso de punteros (el objeto de estudio de la siguiente unidad).

Una lista de **Personas** y **Alumnos**