Programación Orientada a Objetos

Unidad 2: Introducción a la POO
Unidad 3: Relaciones entre clases

Teoría 03 - 01/09/2011 - Pablo Novara

¿Qué es la POO?

La Programación Orientada a Objetos es un paradigma que utiliza objetos como elementos fundamentales en la construcción de la solución.

Objeto = datos/estado (atributos)

+ comportamiento (métodos)

Clase = “tipo” de objeto.

Instancia = un objeto en particular

Objetos en C++

```
class Alumno {
private:
    char nombre[50];
    int *notas, cant_notas;
public:
    void CargarNombre(char *nombre);
    void AgregarNota(int nota);
    void CambiarNota(int cual, int nota);
    const char *VerNombre();
    int VerCantNotas();
    int VerNota(int cual);
    float CalcPromedio();
};
```

Estado (privado, por el principio de ocultación)

Interfaz (lo único que ven los clientes de la clase)

Objetos en C++

```
Alumno a;
a.CargarNombre("Oliver Atom");
a.AgregarNota(5);
a.AgregarNota(4);
a.AgregarNota(8);
float prom = a.CalcPromedio();
cout<<"Promedio: "<<prom<<endl;
```

a es una *instancia* de la clase Alumno

Los métodos se llaman como a funciones, pero anteponiendo el objeto y el punto

Constructores y Destructores

Constructor:

- método que se invoca automáticamente al crear un objeto.
- pueden recibir argumentos y sobrecargarse
- si no se especifica ninguno, C++ otorga por defecto un constructor nulo y un constructor de copia

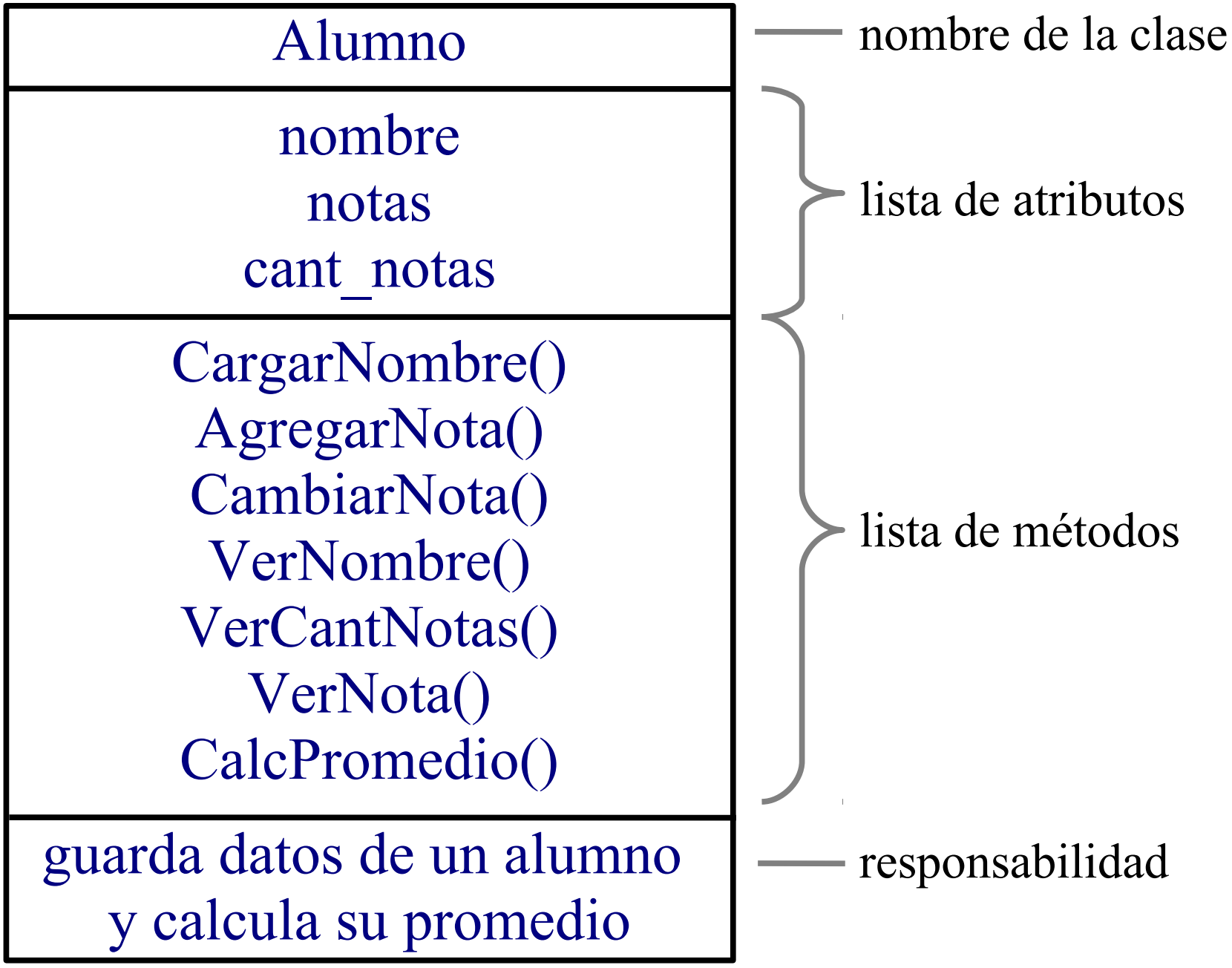
Destructor:

- método que se invoca automáticamente al destruir un objetos.
- no puede recibir parámetros

Constructores y Destructores

```
class Alumno {
    char nombre[50]; int *notas, cant_notas;
public:
    Alumno();  // ctor por defecto, luego se usara
               // CargarNomre para asignar el nombre
    Alumno(char *nombre); // ctor que recibe el nombre
    Alumno(char *nombre, int *notas, int cant); // recibe
                                                // nombre y notas
    Alumno(const Alumno &a); // ctor de copia
    ~Alumno(); // destructor, para liberar la memoria
               // reservada para el arreglo de notas
… };
```

Notación UML (clases)



El puntero this

Dentro de un método existe un puntero predefinido denominado **this** que apunta al objeto (instancia) en el que se encuentra

```
class Complejo {
    int r,i;
  public:
    void Cargar(int r, int i) {
        this->r=r; this->i=i;
    }
    ...
};
```

El puntero this

```
class Complejo {
    ...
    Complejo &Sumar(Complejo e2) {
        r+=e2.r; i+=e2.i; return *this;
    }
    Complejo &Restar(Complejo e2) {
        r-=e2.r; i-=e2.i; return *this;
    }
};

...
// en el main, teniendo instancias e1,e2,e3,e4
    e1.Sumar(e2).Restar(e3).Sumar(e4);
// todo se aplica a e1, que guardara el resultado final
```

el método devuelve el mismo objeto con el que se llamó

permite hacer llamadas en cadena a métodos de un mismo objeto

Métodos y Atributos Estáticos

Un método/atributo estático es aquel que **no está asociado a ninguna instancia** de una clase, sino que es único, **el mismo para todos los objetos**.

Pueden ser accedidos por cualquier instancia de la misma, aunque también se puede acceder a ellos sin construir una instancia de la clase.

Un método estático solo puede acceder a otros métodos/atributos estáticos.

Métodos y Atributos Estáticos

```
class CuentaInstancias {
    static int contador;
public:
    CuentaInstancias() { contador++; }
    ~CuentaInstancias() { contador--; }
    static int VerContador() { return contador; }
};
int CuentaInstancias::contador=0;
```

Métodos y Atributos Estáticos

Los métodos estáticos se pueden invocar con una instancia de la clase o utilizando el operador de scope (::)

```
int main() {
    cout<<CuentaInstancias::VerContador()<<endl;
    CuentaInstancias c1;
    CuentaInstancias *a=new CuentaInstancias[10];
    cout<<c1.VerContador()<<endl;
    delete [] a;
    cout<<CuentaInstancias::VerContador()<<endl;
    return 0;
}
```

Ejemplo Integrador

1) Diseñe una clase Hormiga para representar una hormiga como esta: **oo<** que camina dentro de un rectángulo

2) Escriba un primer programa cliente que genere una hormiga y la muestre caminando por la pantalla.

3) Escriba un segundo programa cliente que genere una colonia de N hormigas utilizando un vector de Hormigas.

Cómo modelar una Hormiga que camina

1) Identificar atributos útiles:

- Propios de cualquier hormiga:
  - posición (¿donde esta?)
  - dirección (¿hacia donde camina?)
  - color (¿negra o colorada?)
- Adicionales para el problema
  - limites (tamaño del rectángulo donde camina, necesario para limitar el movimiento)

Cómo modelar una Hormiga que camina

2) Identificar métodos:

- Constructor (para definir color, posición inicial, etc)
- Avanzar (para que camine un paso)
- CambiarDireccion (para hacer que gire)
- Selectores (para consultar la posición, color, etc.)
- Destructor (no es necesario)

Cómo programar una Hormiga que camina

3) Implementar la clase en C++:

– Hormiga.h:

```
#ifndef HORMIGA_H
#define HORMIGA_H

class Hormiga {
    // ...atributos...
  public:
    // ...metodos (solo prototipos)...
  };

#endif
```

Evita que se declare 2 veces

Cómo programar una Hormiga que camina

3) Implementar la clase en C++:

– Hormiga.cpp:

```
#include “Hormiga.h”

// métodos completos
  void Hormiga::Avanzar() {
    …
  }
  void Hormiga::CambiarDireccion() {
    …
  }
  ...
```

Cómo programar una Hormiga que camina

4) Usar la clase en el programa cliente:

- main.cpp:

```
#include “Hormiga.h”

void dibuja_hormiga(Hormiga &h);

int main() {
    Hormiga h(10,10,1,true);
    while(true) {
        h.Avanzar();
        if (rand()%5==1) h.CambiarDireccion();
        dibuja_Hormiga(&h);
        ...
```