Grado en Ingeniería Información

# PROGRAMACIÓN II - Sesión 8

## Tema 5.

## Herencia

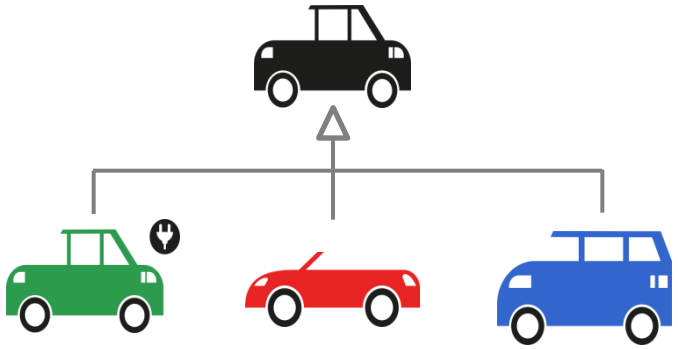
Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

UNIVERSIDAD NEBRIJA

# T5. Herencia

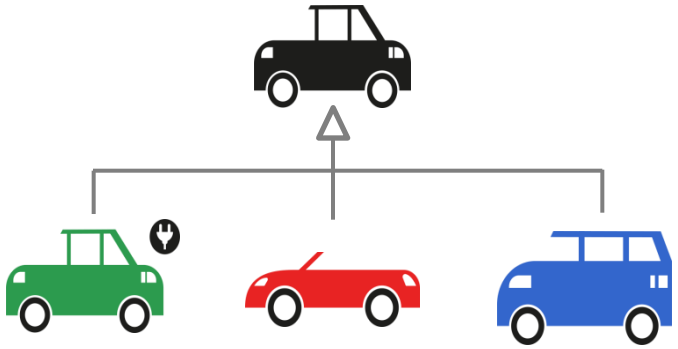




# ÍNDICE

UNIVERSIDAD NEBRIJA

# ÍNDICE



**La POO** permite programar de forma **más eficiente**:

- Mejora la capacidad para analizar un problema.
- Evita la duplicidad, permitiendo usar clases dentro de otras clases.
- Reutiliza código (ya probado) e ideas anteriores.

## COMPOSICIÓN

Es la **creación de objetos** de una **clase ya existente dentro** de **otra clase que se está creando**.

Así, la **nueva clase** está compuesta de **objetos de clases ya existentes**.

# Herencia y Composición

```
class Fecha  {
    private:
        int dd, mm, aa;
    public:
        Fecha():dd(1),mm(1), aa(1900){};
        Fecha(int d, int m, int a);
        void mostrarFecha () const;

class Persona  {
    private:
        string nombre, apellidos;
        Fecha    fnac;   // Composición
    public:
        Persona();
        Persona(string const &nom, string  const  &apel, Fecha const &f);
        void mostrarPersona () const;
```

```
//----- METODOS CLASE FECHA
Fecha::Fecha(int d, int m, int a)  {
    dd = d;
    mm = m;
    aa = a;
}

void Fecha::mostrarFecha () const {
    cout << " " << dd << "/" << mm << "/" << aa;
}


//----- METODOS CLASE PERSONA
Persona::Persona():fnac(1,1,1900) {
    nombre.clear();
    apellidos.clear();
}

Persona::Persona(string const &nom, string  const &apel, Fecha const &f) {
    nombre = nom;
    apellidos = apel;
    fnac = f;
}

void Persona::mostrarPersona () const {
    cout << "\n\n\tNombre: " << getNombre();
    cout << "\n\n\tApellidos: " << getApellidos();
    cout << "\n\tFecha nacimiento: ";
    fnac.mostrarFecha();
}
```

# HERENCIA

Es la **creación de una *clase* (derivada, subclase)** del **tipo de otra *clase* (base, superclase) ya existente**.

Así, la **nueva clase (derivada)** toma la forma de la **clase existente**, heredando sus atributos y métodos, **(base)** y le **añade código**, sin modificar la que ya existía.

Se trata de un pilar fundamental de la POO que **permite**:

- ***Crear nuevas clases a partir de las existentes.***
- ***Conservar las propiedades de la clase original.***
- ***Añadir nuevos métodos y atributos a la clase original.***

- ✓ **HERENCIA SIMPLE** es cuando una **clase deriva** de una **clase base**.
- ✓ **HERENCIA MÚLTIPLE** es cuando una **clase deriva** de **más** de una **clase base**.
- ✓ Una **clase derivada** podrá ser **clase base de otra clase**.

## La HERENCIA se utiliza cuando:

- Existen **suficientes similitudes** entre la **clase derivada** (nueva clase, **subclase**) y la **clase base** (clase existente, **superclase**).
- La **clase derivada:**
  - **Ampliará** o **redefinirá** el conjunto de **características** que posee la **clase base**.
  - Adoptará todos los miembros de la **clase base**: **atributos y métodos**.

## CLASE BASE (superclase):

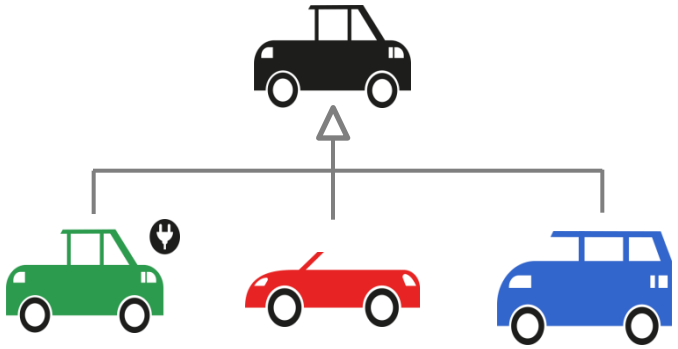
- Es la **clase ya creada** y de la que se **hereda**.
- Clase Madre.
- La herencia **no afecta** a la **Clase Base**.
- Más **genéricas**, más **abstractas**, con **menor nivel de detalle**.
- Contienen los conceptos fundamentales.

# CLASE DERIVADA (subclase):

- Es la **clase que se crea** a partir de la **Clase Base**. Es la clase que **hereda**.
- Clase Hija.
- **Hereda todas las características** de la **Clase Base** y puede **definir nuevas características**.
- Puede **redefinir características** de la **Clase Base** y puede **anular características heredadas**.
- **No puede acceder a los miembros privados** de la **Clase Base**.
- Puede servir de **Clase Base** para **otras clases**.

UNIVERSIDAD NEBRIJA

# ÍNDICE



La herencia permite crear **Jerarquías de Clases**.

Las **Jerarquías de Clases** permiten la resolución de **problemas complejos**.

El **refinamiento de la funcionalidad** (más **general**) de las **Clases Base (superclases)** va creando **Clases Derivadas (subclases)** que presentan un comportamiento más "**especializando**" o **particular**.

La primera **Clase de la Jerarquía**, debe de **implementar todas las características** que son **comunes a todas** las **clases de la jerarquía**.

Cada **subclase** debe contemplar únicamente las **peculiaridades propias que la distinguen** de su **superclase**.

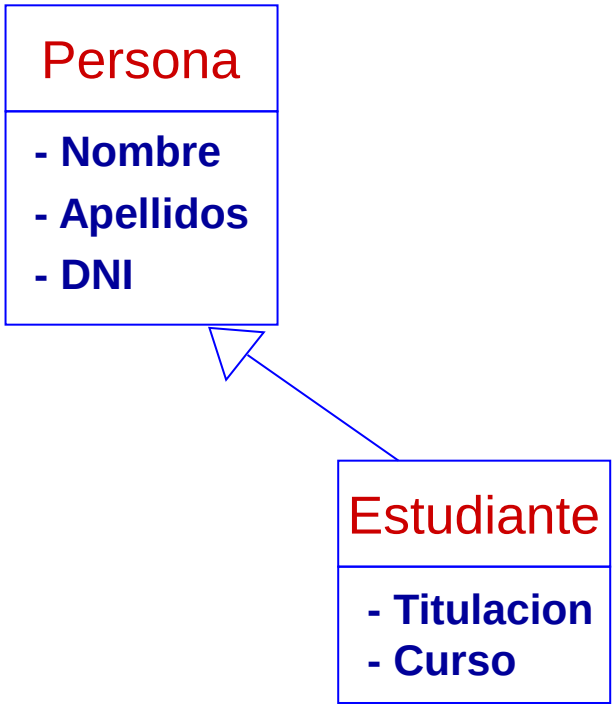
# Ejemplo: Jerarquías de clase

**Ejercicio**: Realizar un programa para los integrantes de un centro educativo.

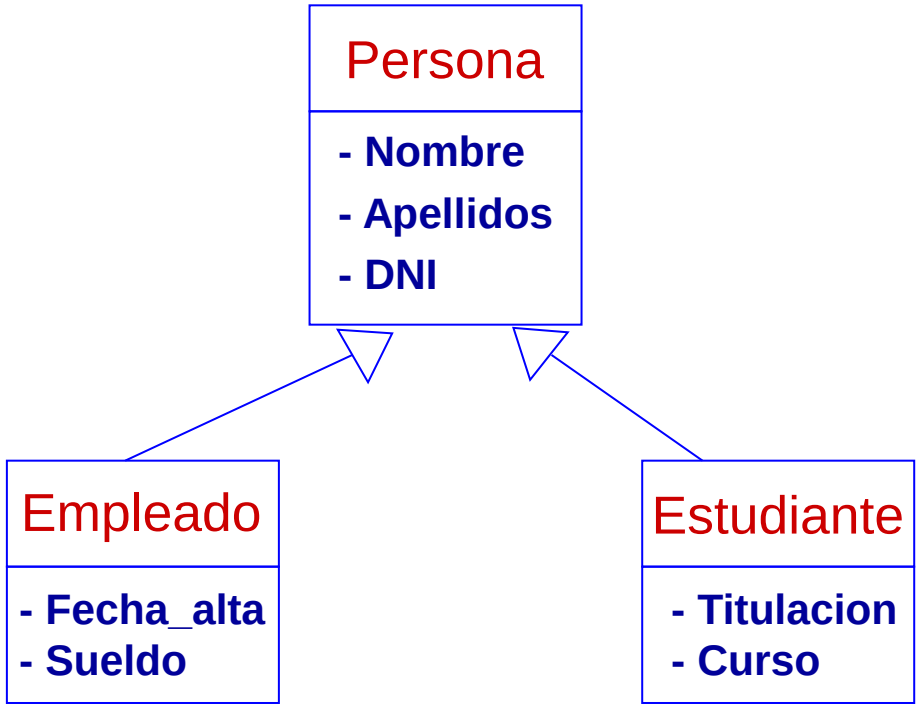
- Estudiantes
- PAS
- PDI


Persona
- Nombre
- Apellidos
- DNI

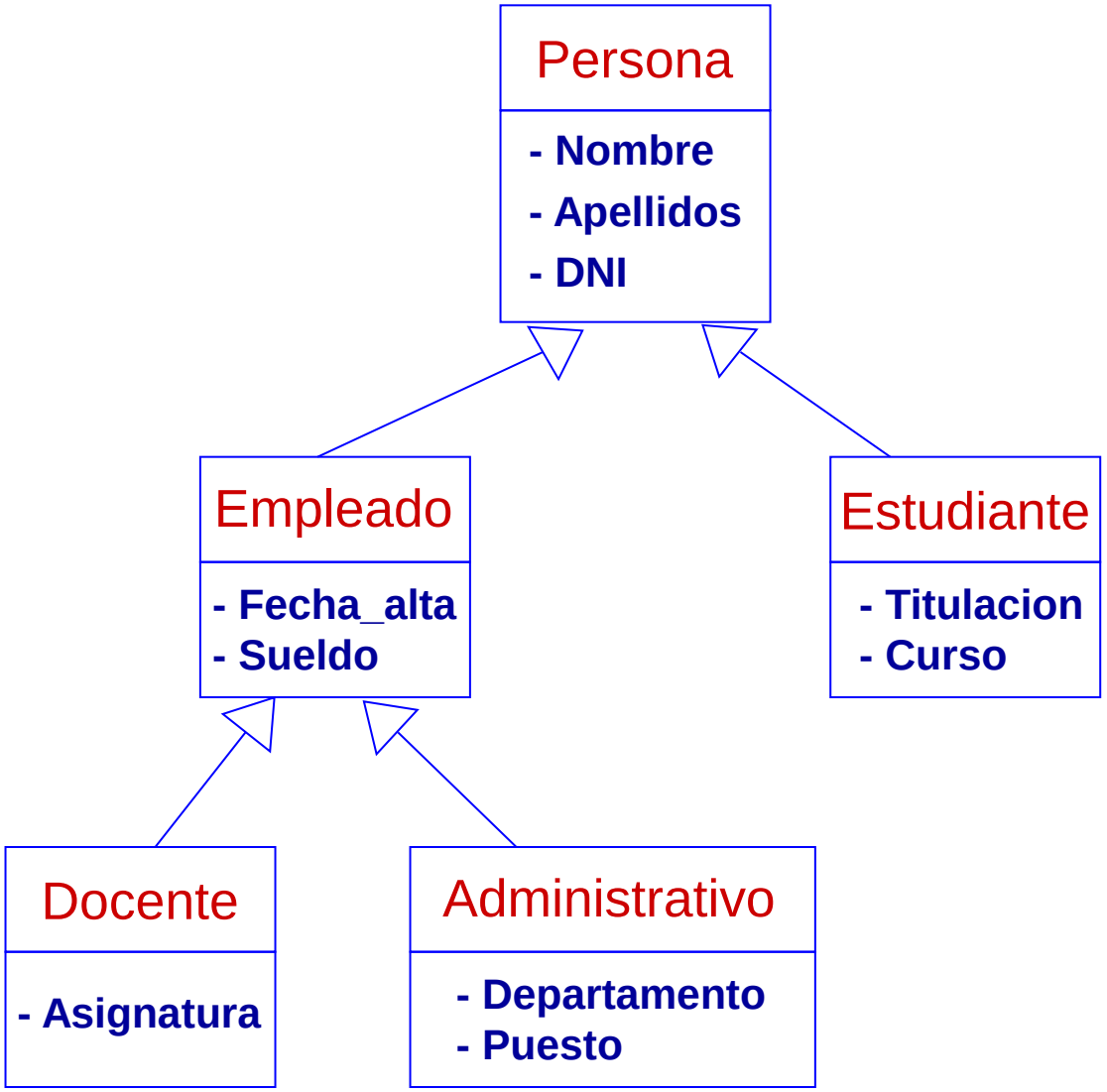
# Jerarquías de clase



# Jerarquías de clase



# Relación *is-a* (*es un/a*)



**Ejercicio**: Realizar un programa para manejar :

- Triángulo equilátero
- Cuadrado
- Pentágono regular
- etc.

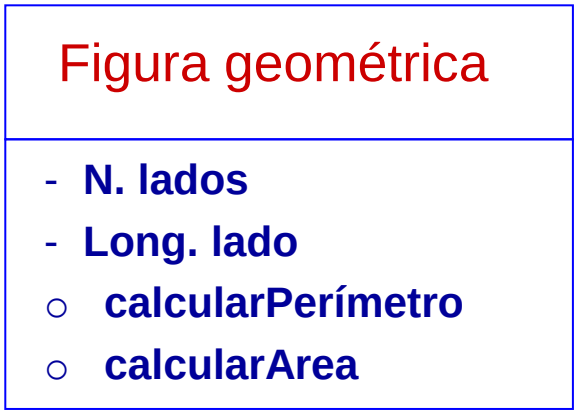
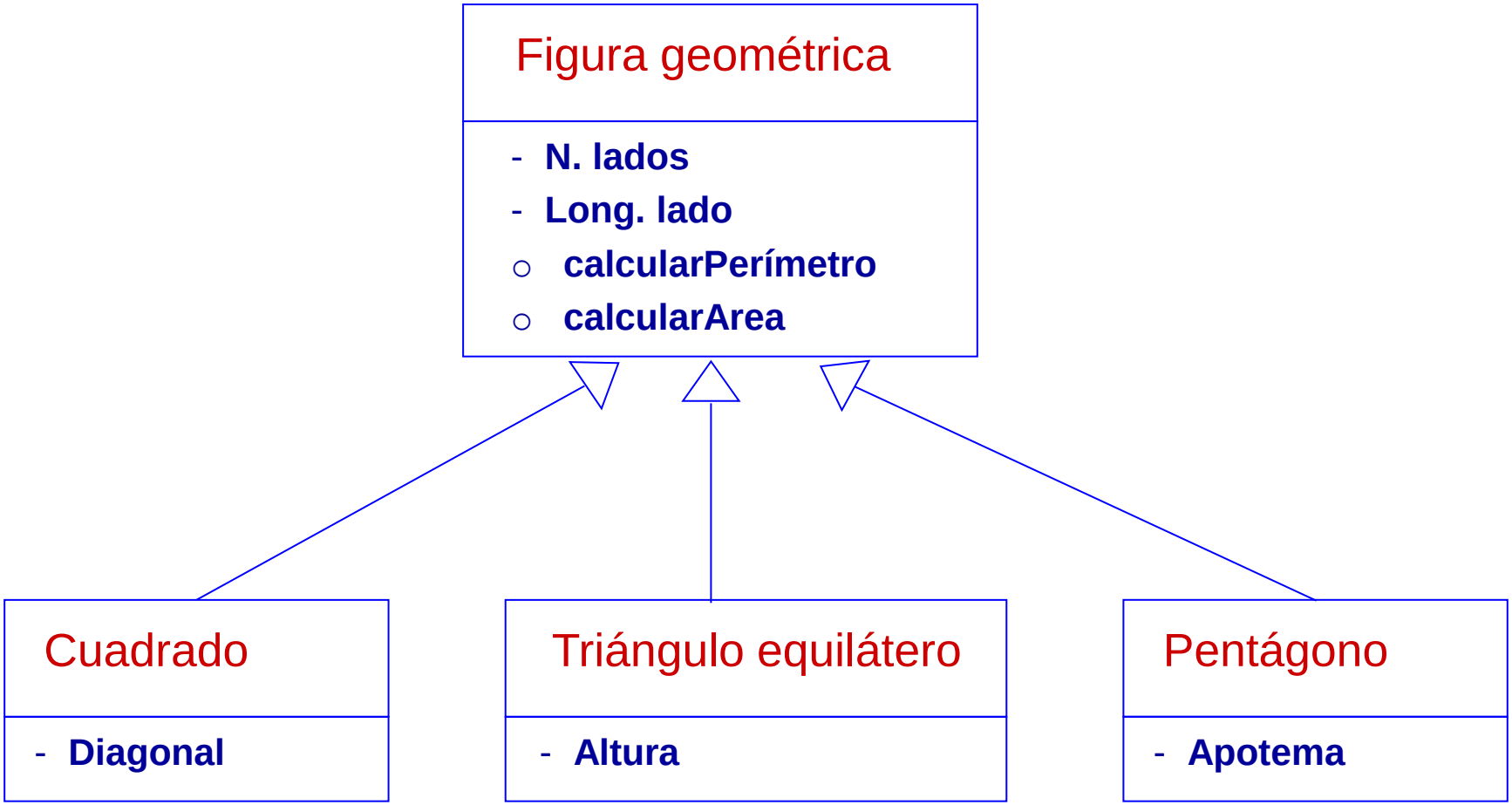


Figura geométrica
- N. lados
- Long. lado
○ calcularPerímetro
○ calcularArea

# Jerarquías de clase



# Reglas de la Herencia

Una subclase hereda todos los métodos y atributos de su clase base o superclase, **excepto los constructores**:

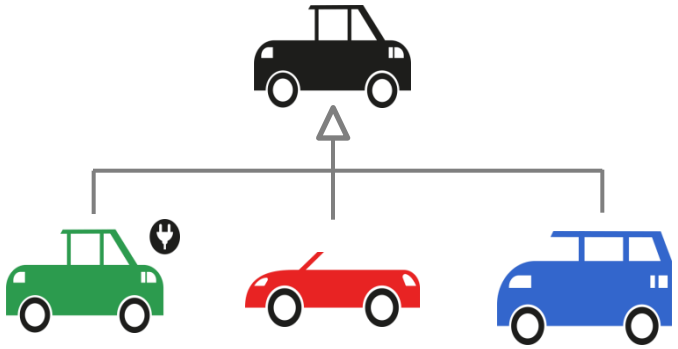
- La clase derivada o subclase **no tiene acceso a variables miembro privadas** de su clase base.
- La clase derivada o subclase **tiene acceso a variables miembro públicas** de su clase base.

Una clase derivada puede añadir sus propios miembros (atributos y métodos). Si los nombres coinciden con los de la clase base, queda oculto para la clase derivada.

Los miembros heredados por la clase derivada pueden ser heredados por otras clases derivadas de ellas (**propagación de herencia**).

UNIVERSIDAD NEBRIJA

# T5. Herencia





# ÍNDICE

**SINTAXIS:**

***class <clase_derivada>: [public/private/protected]***

***<clase_base1>*** *[,[public/private/protected] <clase_base2>] {};*

Control de Acceso en la herencia a la Clase Base:

- ***public***
- ***private***
- ***protected***

```
class figuraGeometrica {
    private:
        int nlados;
        float longitud;
    public:
        figuraGeometrica():nlados(0), longitud(0){}
        figuraGeometrica (int n, float l):nlados(n), longitud(l){}
        float calcularPerimetro() const;
        void setLongitud(float l);
        float getLongitud() const;
        void setNLados(int n);
};
```

```
class triangulo: public figuraGeometrica {
    private:
        float altura;
    public:
        triangulo():altura(0) {}
        void setAltura(float h);
        float getAltura() const;
        void calcularAltura();
};
```

```
// Clase FIGURA GEOMETRICA
float figuraGeometrica::calcularPerimetro() const {
    return nlados*longitud;
}

void figuraGeometrica::setLongitud(float l) {
    longitud = l;
}

float figuraGeometrica::getLongitud() const {
    return longitud;
}
void figuraGeometrica::setNLados(int n) {
    nlados = n;
}
```

```
// Clase TRIANGULO
void triangulo::setAltura(float h) {
    altura = h;
}

float triangulo::getAltura() const {
    return altura;
}

void triangulo::calcularAltura() {
    altura = sqrt(pow(getLongitud(),2)
                  - pow(getLongitud()/2,2));
}
```

```
int main()
{
    triangulo T1;

    T1.setLongitud(4);
    T1.setNLados(3);
    cout << "\n\n\tEl perimetro del triangulo es: "
         << T1.calcularPerimetro();
    T1.calcularAltura();
    cout << "\n\n\tLa altura del triangulo es: "
         << T1.getAltura();
```

**Ejercicio**: Incorporar nuevas clases:

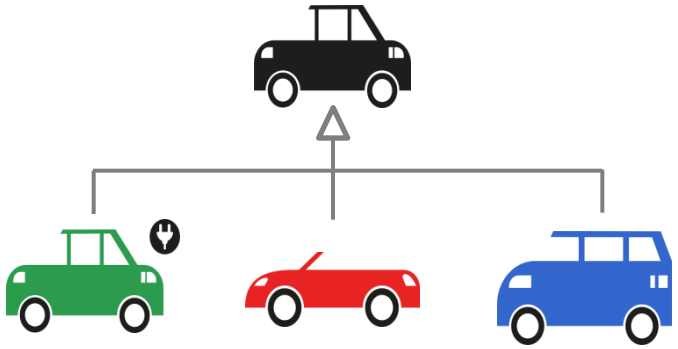
- Cuadrado, para calcular y guardar la diagonal
- Pentágono regular
- etc.

```
class cuadrado: public figuraGeometrica {
    private:
        float diagonal;
    public:
        cuadrado():diagonal(0) {}
        void setDiagonal(float d);
        float getDiagonal() const;
        void calcularDiagonal();
};
```

```
    cuadrado C1;
    C1.setLongitud(2);
    C1.setNLados(4);
    cout << "\n\n\tEl perimetro del cuadrado es: "
         << C1.calcularPerimetro();
    C1.calcularDiagonal();
    cout << "\n\n\tLa diagonal del cuadrado es: "
         << C1.getDiagonal();


    cout << "\n\n\t";
    return 0;
}
```

UNIVERSIDAD NEBRIJA

# T5. Herencia





# ÍNDICE

# Modificadores de acceso dentro de la clase

Los **atributos** y los **métodos** de una clase son **siempre privados**, **salvo que se indique otra cosa**.

Los **modos de acceso** controlan **quién accede a los atributos o métodos de la clase**:

- ***private*** Solo los objetos de la **propia clase o de una clase amiga** (***friend***) pueden acceder a los atributos y métodos. Esta parte de la clase corresponde, normalmente, a la **implementación** de la clase.
- ***public*** **Cualquier objeto de la aplicación puede acceder** a los atributos y métodos así definidos. Esta parte define la **interfaz** de la clase.
- ***protected*** Acceden a los atributos y métodos sólo los objetos de la **propia clase, las clases derivas de ella y clases amigas** (***friend***).

# Modificadores de acceso dentro de la clase

**Acceso** desde **diferentes partes de la aplicación** a los **atributos** y **métodos** declarados en cada uno de los tres posibles bloques de una **clase**.

| Acceso desde / Bloque | **funciones miembro** | clases ***amigas*** | clases ***derivadas*** | otras **clases** o **exterior** |
|---|---|---|---|---|
| ***public*** | **Si** | **Si** | **Si** | **Si** |
| ***private*** | **Si** | **Si** | **No** | **No** |
| ***protected*** | **Si** | **Si** | **Si** | **No** |

# Modificadores para el control de acceso en la herencia

Los **modos de acceso** controlan el **acceso en la herencia**, **quién accede a los atributos o métodos de la clase base** :

- ***private*** Los miembros (atributos y métodos) **públicos** o **protegidos** de la **clase base** son **privados** para la **clase derivada**.

  Los miembros **privados** de la **clase base no son accesibles** desde la **clase derivada**.

- ***public*** Los miembros heredados de la **clase base mantienen el tipo de acceso** con que fueron declarados en la **clase derivada**.

- ***protected*** Los miembros (atributos y métodos) **públicos** o **protegidos** de la **clase base** son **protegidos** para la **clase derivada**.

  Los miembros **privados** de la **clase base no son accesibles** desde la **clase derivada**.

# Control de acceso en la herencia

| Nivel de acceso declarado en la Clase Base | *Público* | *Protegido* | *Privado* |
|---|---|---|---|
| Nivel de acceso resultante en la **Clase Derivada** (modificador ***public***) | **Público** | **Protegido** | **Privado** |
| Acceso desde el **resto de la aplicación** a través de la **Clase Derivada** (modificador ***public***) | **SI** | **NO** | **NO** |
| Nivel de acceso resultante en la **Clase Derivada** (modificador ***private***) | **Privado** | **Privado** | **Sin acceso** |
| Acceso desde el **resto de la aplicación** a través de la **Clase Derivada** (modificador ***private***) | **NO** | **NO** | **NO** |
| Nivel de acceso resultante en la **Clase Derivada** (modificador ***protected***) | **Protegido** | **Protegido** | **Sin acceso** |
| Acceso desde el **resto de la aplicación** a través de la **Clase Derivada** (modificador ***protected***) | **NO** | **NO** | **NO** |

# Control de acceso en la herencia

## Modificador public

```
class CBase
{     private:
            int b1;
            int b2;
      public:
            int b3;
            void metodoB1(int a);
            void metodoB2 ();
};
class CDerivada : public CBase
{     private:
            float f1;
            char c1;
      public:
            float metodoD1();
            void metodoD2 (char c);
};
```

# Control de acceso en la herencia

## *Modificador public*

```
CBase  ob1, ob2;

CDerivada  od1, od2;

ob1.metodoB2();

ob2.b3 = 6;

cout << od1.metodoD1();

od2.metodoB2();

od1.b3 = 3;

// ERROR, no se puede acceder a un atributo privado de la clase base

od1.b1 = 15;
```

# Control de acceso en la herencia

## Modificador public

```
class CBase
{     protected:
          int b1;
          int b2;
      public:
          int b3;
          void metodoB1(int a);
          void metodoB2 ();
}
class CDerivada : public CBase
{     private:
          float f1;
          char c1;
      public:
          float metodoD1();
          void metodoD2 (char c);
}
```

Modificador public

```
void metodoD2 (char c)
{
    f1 = 8.5;
    c1 = c;
    // Acceso PERMITIDO desde la clase derivada
    // por ser atributos PROTECTED en la clase base
    b1 = 10;
    b2 = 15;
    b3 = 20;
}
```

# Control de acceso en la herencia

## *Modificador public*

```
using namespace std;

// CLASE PERSONA
class Persona
{    private:
          string apellidos;
     protected:
          string nombre;
     public:
          string iniciales;

          Persona ();
          Persona (const string &n, const string &a, const string &i);

          string getApellidos () const {return apellidos;}
          string getNombre() const {return nombre;}  // 1era. opción
};
```

# Control de acceso en la herencia

## *Modificador public*

```
//CLASE EMPLEADO
class Empleado : public Persona
{  private:
          int sueldo;
    public:
          Empleado (const string &n, const string &a, const string &i, int s);
          int getSueldo() const {return sueldo; }
//        string getNombre() const {return nombre;}  // 2da. opción
};
```

# Control de acceso en la herencia

## *Modificador public*

```
int main()
{
    Empleado trabajador ("Manuel.", "Garcia Sanchez", "MGS", 2000);

    cout << "\n\n\tEl salario de " << trabajador.getNombre() << " ";

    cout << trabajador.getApellidos() << " -- "
         << trabajador.iniciales;

    cout << " es de " << trabajador.getSueldo() << " euros\n\n";

    cout << "\n\n\t";
    return 0;
}
```

# Control de acceso en la herencia

## *Modificador public*

```
//---------------CLASE PERSONA---------------------
Persona::Persona(){
     nombre="";
     apellidos="";
     iniciales="";
}

Persona::Persona(const string &n, const string &a,
                 const string &i) {
     nombre = n;
     apellidos = a;
     iniciales = i;
}

//---------------CLASE EMPLEADO---------------------
Empleado::Empleado (const string &n, const string &a,
                    const string &i, int s):Persona(n, a, i) {
     sueldo = s;
}
```

# Control de acceso en la herencia

## *Modificador private*

```
//CLASE EMPLEADO
class Empleado : private Persona
{   private:
          int sueldo;
    public:
        Empleado (const string &n, const string &a, const string &i, int s);
        int getSueldo() const {return sueldo; }

        string getNombre() {return nombre;}
        string getIniciales() {return iniciales;}
        string getApellidosEmpleado(){return getApellidos();}
};
```

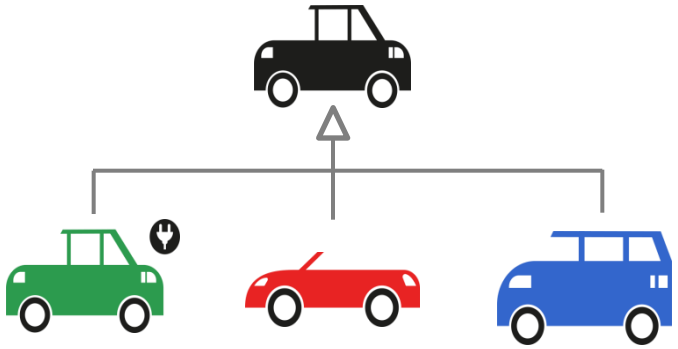
# Control de acceso en la herencia

## *Modificador private*

```
int main()
{
    Empleado trabajador ("Manuel.", "Garcia Sanchez", "MGS", 2000);

    cout << "\n\n\tEl salario de " << trabajador.getNombre() << " ";

    //cout << trabajador.getApellidos() << " -- "
    //          << trabajador.iniciales;
    cout << trabajador.getApellidosEmpleado() << " -- "
              << trabajador.getIniciales();

    cout << " es de " << trabajador.getSueldo() << " euros\n\n";

    cout << "\n\n\t";
    return 0;
}
```

UNIVERSIDAD NEBRIJA

# T5. Herencia



5.1. Herencia y Composición

5.2. Jerarquía de clases

5.3. Sintaxis en C++

5.4. Control de acceso en la herencia

**5.5. Miembros que no se hereden automáticamente**

5.6. Redefinición de métodos

5.7. Ejecución en los pases de mensajes con herencia

5.8. Herencia múltiple

# ÍNDICE

# Miembros que no se heredan automáticamente

## NO se HEREDAN:

- Constructores.
- Destructores.
- El Operador de asignación (=) sobrecargado.
- Las funciones/clases ***friend***.

# Miembros que no se heredan automáticamente

## NO se HEREDAN:

- **Constructores.**
- **Destructores.**
- El Operador de asignación (=) sobrecargado.
- Las funciones/clases ***friend***.

# Miembros que no se heredan automáticamente

- Las clases derivadas no heredan los constructores ni los destructores de la clase base.
- Los miembros de la clase base deben ser inicializados:
  - Si hay parámetros, el constructor de la clase base debe ser llamado de forma explícita por el constructor de la clase derivada utilizando los parámetros para la inicialización.
  - Si no hay parámetros (constructor por defecto) no es necesario hacer la llamada de forma explícita.

## ¿Cuál es el orden de ejecución de los métodos constructores y destructores en las clases derivadas?

**Al instanciar una clase derivada:**

1.- Se ejecuta el **constructor** de la **clase base**

2.- Después, se ejecuta el **constructor** de la **clase derivada**

**Al terminar el ciclo de vida de dicha clase:**

1.- Se ejecuta el **destructor** de la **clase derivada**

2.- Después, se ejecuta el **destructor** de la **clase base**

## Miembros que no se heredan automáticamente

## Secuencia de ejecución válida para toda la jerarquía de clases

### Al instanciar una clase derivada:

1.- Se ejecuta el **constructor** de la **clase base**

2.- Si existen objetos miembros, sus constructores se ejecutan después del constructor de la clase base.

3.- Por último, se ejecuta el **constructor** de la **clase derivada**

### Al terminar el ciclo de vida de dicha clase, los destructores se llaman en orden inverso a la derivación:

1.- Se ejecuta el **destructor** de la **clase derivada**

2.- Después los destructores de los objetos miembros.

2.- Finalmente, se ejecuta el **destructor** de la **clase base**

## Cuando se crea un objeto de la clase derivada:

1. Se invoca al **constructor de la clase base**.

   La llamada al constructor de la clase base se realiza con los parámetros que se especifiquen.

   Si no hay parámetros, se usa el constructor por defecto.

2. Se ejecuta el **constructor de la clase derivada**.

# Miembros que no se heredan automáticamente

```
//CLASE EMPLEADO
class Empleado : private Persona
{   private:
          int sueldo;
    public:
 //         Empleado (const string &n, const string &a, const string &i, int s);
        Empleado (const string &n, const string &a,
                  const string &i, int s):Persona(n, a, i), sueldo (s) {}

        int getSueldo() const {return sueldo; }
```

# Miembros que no…: Orden de llamada de Constructores y Destructores

**Cuando se destruye un objeto de la clase derivada:**

1. Se invoca al **destructor de la clase derivada**.
2. Se ejecuta el **destructor de la clase base**.

# Miembros que no se heredan: Constructores y Listas de inicialización

```
class CBase {
    private:
        int b1;
        int b2;
    public:
        int b3;
        CBase();
        CBase(int n1, int n2, int n3);
};

class CDerivada: public CBase {
    private:
        float f1;
        char c1;
    public:
        CDerivada (float d, char e);
        CDerivada (int a, int b, int c, float d, char e);
};
```

# Miembros que no se heredan: Constructores y Listas de inicialización

```
CBase::CBase():b1(0), b2(1), b3(2) {
     cout << "\n\tConstructor CBase- b1: "
          << b1 << " b2: " << b2 << " b3: " << b3;
}

CBase::CBase (int n1, int n2, int n3) {
     b1 = n1;
     b2 = n2;
     b3 = n3;
     cout << "\n\tConstructor CBase- b1: "
          << b1 << " b2: " << b2 << " b3: " << b3;
}
```

# Miembros que no se heredan: Constructores y Listas de inicialización

```
CDerivada :: CDerivada (float d, char e) {
     f1 = d;
     c1 = e;
     cout << "\n\tConstructor CDerivada- f1: "
          << f1 << " c1: " << c1;
}

CDerivada :: CDerivada (int a, int b, int c, float d, char e):
                         CBase(a, b, c) {
     f1 = d;
     c1 = e;
     cout << "\n\tConstructor CDerivada- f1: "
          << f1 << " c1: " << c1;
}
```

# Miembros que no se heredan: Constructores y Listas de inicialización

```
int main() {

    cout << "\n\tOBJETO od1";
    CDerivada od1 (5.2, 'm');

    cout << "\n\n\n\tOBJETO od2";
    CDerivada od2 (11, 22, 33, 15.25, 'k');

    cout << "\n\n\t";
    return 0;
}
```

# Miembros que no se heredan: Constructores y Listas de inicialización

**¿Cómo se ejecuta la siguiente llamada?**

***CDerivada od (5.2, 'm');***

# Miembros que no se heredan: Constructores y Listas de inicialización

**¿Cómo se ejecuta la siguiente llamada?**

***CDerivada od (5.2, 'm');***

1. Se ejecuta el constructor por defecto de la **clase base**.

   Entonces: ***od.b1 = 0; od.b2=1; od.b3=2;***

2. Se ejecuta el constructor de la **clase derivada**.

   Entonces: ***od.f1 = 5.2; od.c1='m';***

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ahora la llamada sería:**

*CDerivada od (11, 22, 33, 15.25, 'k');*

**¿Cómo se ejecutaría?**

## Ahora la llamada sería:

***CDerivada od (11, 22, 33, 15.25, 'k');***

1. Se ejecuta el constructor de la **clase base** con los valores indicados en la llamada.

   Entonces: ***od.b1 = 11; od.b2 = 22; od.b3 = 33;***

2. Se ejecuta el constructor de la **clase derivada**.

   Entonces: ***od.f1 = 15.25; od.c1='k';***

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ejercicio**: ¿Cómo se ejecutan los constructores y qué se muestra en pantalla?

```
class Padre{
 public:
 Padre(int N){cout << "\n\tConstructor Padre "<< N <<endl;}
};
class Hija: public Padre{
 public:
 Hija():Padre(100){cout << "\n\tConstructor Hija"<<endl;}
};
class Nieta: public Hija{
 public:
 Nieta(){cout<< "\n\tConstructor Nieta"<<endl;}
};

int main()
{
    Nieta miClase;
    cout << "\n\n\t";
    return 0;
}
```

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ejercicio**: ¿Cómo se ejecutan los constructores y qué se muestra en pantalla?

```
class Padre{
 public:
 Padre(int N){cout << "\n\tConstructor Padre "<< N <<endl;}
};
class Hija: public Padre{
 public:
 Hija():Padre(100){cout << "\n\tConstructor Hija"<<endl;}
};
class Nieta: public Hija{
 public:
 Nieta(){cout<< "\n\tConstructor Nieta"<<endl;}
};

int main()
{
    Nieta miClase;
    cout << "\n\n\t";
    return 0;
}
```

```
Constructor Padre 100

Constructor Hija

Constructor Nieta
```

UNIVERSIDAD NEBRIJA

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ejercicio**: ¿Cómo se ejecutan los destructores y qué se muestra en pantalla?

```
class Padre {
    public:
        Padre(){ cout << "\n\n\tConstructor Padre";}
        ~Padre(){std::cout << "\n\tDestructor Padre\n\n\t";}
};


class Hija: public Padre {
    public:
        Hija(){std::cout << "\n\n\tConstructor Hija";}
        ~Hija(){std::cout << "\n\tDestructor Hija\n\n\t";}
};


class Nieta: public Hija {
    public:
        Nieta(){cout<< "\n\n\tConstructor Nieta";}
        ~Nieta(){cout<< "\n\tDestructor Nieta\n\n\t";}
};
```

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ejercicio**: ¿Cómo se ejecutan los destructores y qué se muestra en pantalla?

```
int main() {
    shared_ptr<Nieta> miPunt;
    cout << "\n\tEl puntero no direcciona nada.\n";

    miPunt = make_shared<Nieta>(); // Se inicializa el puntero
    cout << "\n\n\n\tEl programa hace sus tareas.";
    cout << "\n\n\tFin del proceso.\n\n";

    return 0;
}
```

# Miembros que no se heredan: Constructores y Listas de inicialización

**Ejercicio**: ¿Cómo se ejecutan los destructores y qué se muestra en pantalla?

```
int main() {
    shared_ptr<Nieta> miP
    cout << "\n\tEl punte

    miPunt = make_shared<                    l puntero
    cout << "\n\n\n\tEl p
    cout << "\n\n\tFin de

    return 0;
}
```

```
El puntero no direcciona nada.


Constructor Padre

Constructor Hija

Constructor Nieta


El programa hace sus tareas.

Fin del proceso.


Destructor Nieta


Destructor Hija


Destructor Padre
```

UNIVERSIDAD NEBRIJA

**Ejercicio**: Realizar un programa con tres clase:

- Clase abuelo:
  - 3 atributos de tipo int: private, protected y public
  - Método: void **iniciarTodo**();
- Clase padre con diferentes accesos de herencia a la clase abuelo: private, protected y public y el método iniciarTodo para dar valor a los atributos.

- Clase nieto con acceso de herencia public a la padre y el método iniciarTodo para dar valor a los atributos.

El main debe declarar un objeto para cada clase. Invocar al método iniciarTodo y mostrar por pantalla el valor que toman los atributos.