Programación Orientada a Objetos

POO

DEFINICION

- La programación orientada a objetos no debe confundirse con un lenguaje programación orientado a objetos.
- La POO es un paradigma, es otra forma de pensar, es una filosofía única que permite solucionar problemas reales mediante la abstracción de los diferentes agentes, entidades o elementos que actúan en el planteamiento de un problema.
- Un Lenguaje de Programación Orientado a Objetos permite hacer uso de este paradigma.
- La POO intenta llevar al mundo del código lo mismo que encontramos en el mundo real.
- Al mirar alrededor vemos cosas, **objetos** que podemos reconocer porque cada uno pertenece a una **clase**. Podemos distinguir un caballo de un perro porque son de clases diferentes.

DEFINICION

- Ejemplo:
  a. **Problema:** Una persona necesita ver televisión.
  b. **Solución:** Existen 3 elementos o agentes que se pueden abstraer del problema:

- En el problema planteado se especifican 3 elementos involucrados. Cada elemento posee sus propias características y sus propios comportamientos. En POO a estos elementos se les conoce bajo el nombre de **OBJETOS**.

- En POO a las características que identifican a cada objeto se le denominan **ATRIBUTOS** y a los comportamientos se les denominan **METODOS**.

ELEMENTO	DESCRIPCION
Persona	Tiene sus propios atributos: Color piel, Altura, genero, Color ojos, Cabello, etc. Y tiene un comportamiento: Ver , escuchar, hablar, etc.
Control Remoto	Tiene sus propios atributos: Tamaño, color, tipo, batería, etc. Y tiene un comportamiento: Enviar señal, codificar señal, cambiar canal, aumentar volumen, ingresar a menú, prender TV etc.
Televisor	Tiene sus propios atributos: pulgadas, tipo, numero parlantes, marca , etc. Y tiene un comportamiento: Decodificar señal, prender, apagar, emitir señal, emitir audio, etc.

DEFINICION DE CLASE

- Una **CLASE** es una plantilla mediante la cual se crean los diferentes objetos requeridos para la solución del problema. Los Objetos son instancias de las clases.

- Las clases son a los objetos como los tipos de datos son a las variables.

- Ejemplo: Se puede crear un objeto llamado Cesar. Este objeto es creado a partir de la clase Persona. Se puede crear otro objeto llamado: Patricia el cual pertenece a la clase Persona. Significa que a partir de la clase se pueden crear los objetos que se deseen.

- Ejemplo: Se puede crear un objeto llamado LCD LG, el cual pertenece a la clase Televisor.

ELEMENTO	DESCRIPCION
Persona	Tiene sus propios atributos: Color piel, Altura, genero, Color ojos, Cabello, etc. Y tiene un comportamiento: Ver , escuchar, hablar, etc.
Control Remoto	Tiene sus propios atributos: Tamaño, color, tipo, batería, etc. Y tiene un comportamiento: Enviar señal, codificar señal, cambiar canal, aumentar volumen, ingresar a menú, prender TV etc.
Televisor	Tiene sus propios atributos: pulgadas, tipo, numero parlantes, marca , etc. Y tiene un comportamiento: Decodificar señal, prender, apagar, emitir señal, emitir audio, etc.

DEFINICION DE OBJETO

- Es una instancia de una clase. Por lo tanto, los objetos hacen uso de los ***Atributos*** (variables) y ***Métodos*** (Funciones y Procedimientos) de su correspondiente Clase.

- Es una variable de tipo clase. *Por ejemplo*: El objeto Cesar es un objeto de tipo Clase: **Persona**.

- Permiten modelar entidades del mundo real. Por ejemplo: LCD LG pertenece a la clase Televisor. Resumiendo la clase televisor seria:

ATRIBUTOS
tipo. De tipo cadena. **Resolución.** De tipo cadena **Marca.** De tipo cadena.
METODOS
Emitir_Señal ( ) Emitir_Audio ( ) Decodificar_Señal (señal_entrada)

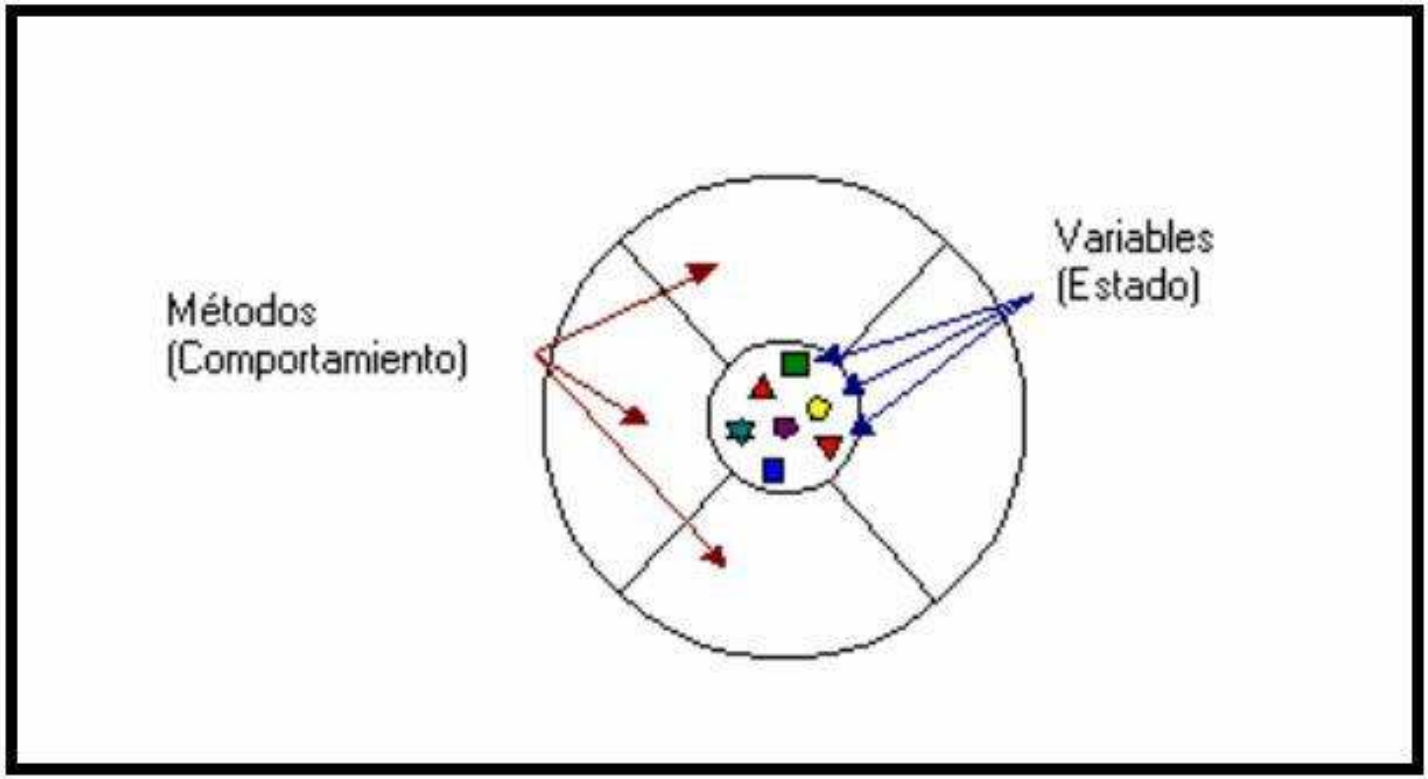
DEFINICION DE OBJETO

- Como se puede observar un objeto a través de su clase esta compuesto por 2 partes: Atributos o propiedades y Métodos que definen el comportamiento de dicho objetos a partir de sus atributos.

- Los atributos y los métodos pueden ser o no accedidos desde afuera dependiendo de la solución a plantear. Por lo general los atributos siempre se ocultan al exterior y algunos métodos quedan visibles al exterior para convertirse en la interfaz del objeto. **Encapsulamiento.**

ATRIBUTOS
tipo. De tipo cadena. **Resolución.** De tipo cadena **Marca.** De tipo cadena.
METODOS
Emitir_Señal ( ) Emitir_Audio ( ) Decodificar_Señal (señal_entrada)

DEFINICION DE OBJETO



IDENTIFICACION DE OBJETOS

- La primera tarea a la que se enfrenta un diseñador o programador en POO es la identificación e los objetos inmersos en el problema a solucionar.

- Los objetos generalmente se ubican en las siguientes categorías:

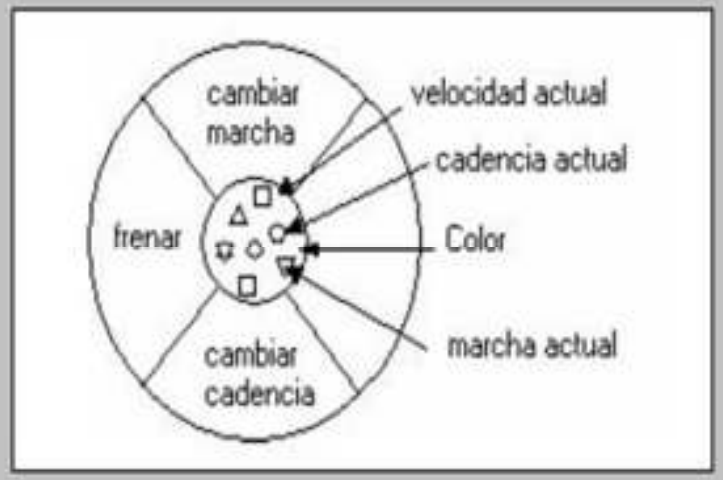
  - Cosas Tangibles: Avión, auto, producto, insumo.
  - Roles : gerente, cliente, vendedor, auxiliar, empleado.
  - Organizaciones o entidades: Empresa, colegio, proveedor, EPS.
  - Cosas intangibles: Vuelos, Servicios, Materias, programas.

EJEMPLO DE OBJETO

OTRO EJEMPLO:

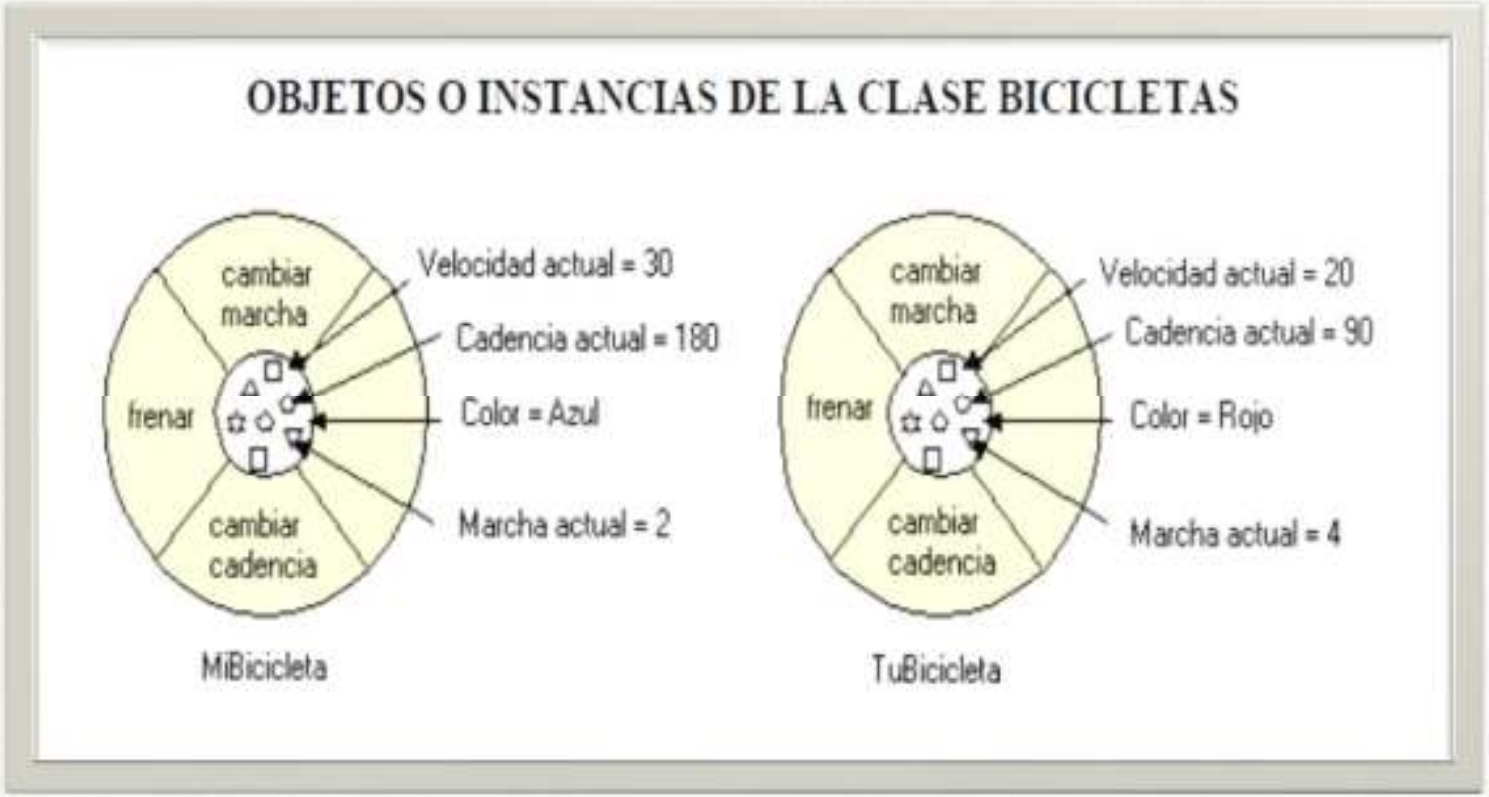
- Se pretende modelar un objeto llamado CARRO el cual existe en el mundo real. Este objeto tiene unos atributos o variables: ***Vel_Max, Color, No_chasis, No_puertas, No._llantas***, ***tipo***. Unos comportamientos y métodos: ***Acelerar*** *(velocidad)*, ***Frenar*** *(velocidad)*, ***mover_cambio*** *(No_cambio)*,

EJEMPLO CLASE



Definición de la clase "bicicleta"

EJEMPLO OBJETOS



CARACTERISTICAS IMPORTANTES DE LA POO

- ABSTRACCION.
- ENCAPSULAMIENTO.
- MENSAJES.
- POLIMORFISMO.
- HERENCIA.

ABSTRACCION

- Es una de las principales características a tener en cuenta ya que permite vislumbrar los diferentes agentes u objetos implicados en un problema.

- La abstracción es la capacidad de aislar un elemento de su contexto para poder verlos de forma singular. Se hace énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?"

- Un ejemplo de abstracción es el cuerpo humano. El cuerpo es una unidad que a su vez está dividido en sistemas (respiratorio, linfático, cardiovascular, etc.), mismos que están divididos en elementos más pequeños: los órganos, y así sucesivamente.

ABSTRACCION

- La abstracción es una estrategia de resolución de problemas en la cual el programador se concentra en resolver una parte del problema *ignorando* completamente los detalles sobre cómo se resuelven el resto de las partes. (*Divide y vencerás*).

- En este proceso de abstracción se considera que el resto de las partes ya han sido resueltas y por lo tanto pueden servir de apoyo para resolver la parte que recibe la atención.

- La abstracción es la estrategia de programación más importante en computación. Sin abstracción las personas serían incapaces de abordar los problemas complejos.

- La pericia de un programador no está en ser veloz para escribir líneas de programa, si no que en saber descubrir, en el proceso de diseño, cuáles son las partes del problema, y luego resolver cada una de ellas abstrayéndose de las otras.

ABSTRACCION

- Un ejemplo de abstracción es el hecho de que uno pueda conducir un automóvil sin ser un mecánico (lo cual probablemente no era cierto con los primeros vehículos). Al conducir, uno se abstrae de cómo funciona la combustión en el motor. Sólo se requiere saber cómo se maneja el volante y los pedales, y cuales son las reglas del tránsito.

ENCAPSULAMIENTO

- Esta propiedad permite el ocultamiento de la información, es decir, permite asegurar que el contenido de un objeto se pueda ocultar del mundo exterior dejándose ver lo que cada objeto necesite hacer publico.

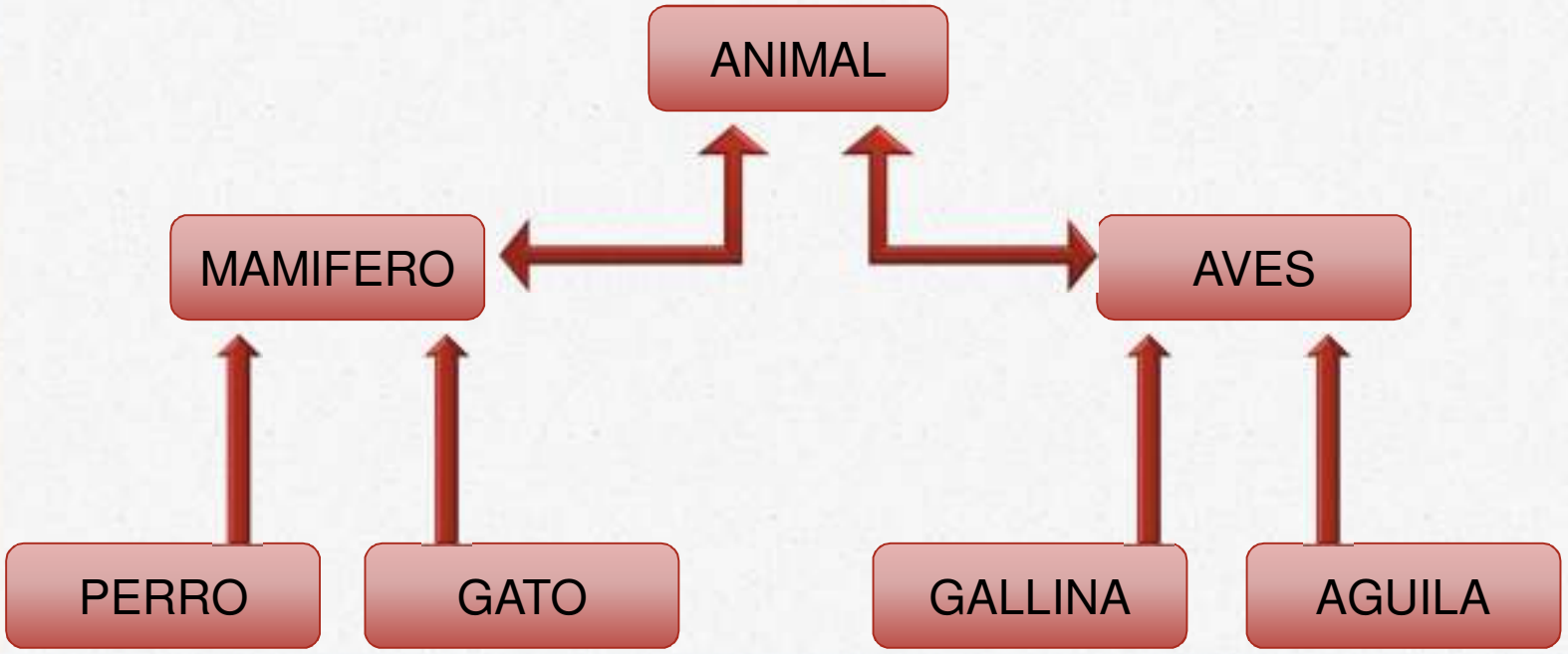
- Por ejemplo: una televisión.

- Tiene 2 ventajas:

  - El usuario puede ser controlado internamente, incluso sus errores, evitando que todo colapse por una intervención indeseada.
  - Si el código está oculto se pueden hacer cambios o mejoras sin que eso afecte el modo como los usuarios van a utilizar el código, únicamente hay que mantener igual la forma de acceder a él, es decir, la interfaz.

HERENCIA

- El mecanismo de herencia permite definir nuevas clases partiendo de otras ya existentes. Las clases que derivan de otras heredan automáticamente todo su comportamiento, pero además pueden introducir características particulares propias que las diferencian.
- La principal ventaja de la herencia es evitar escribir el mismo código varias veces.

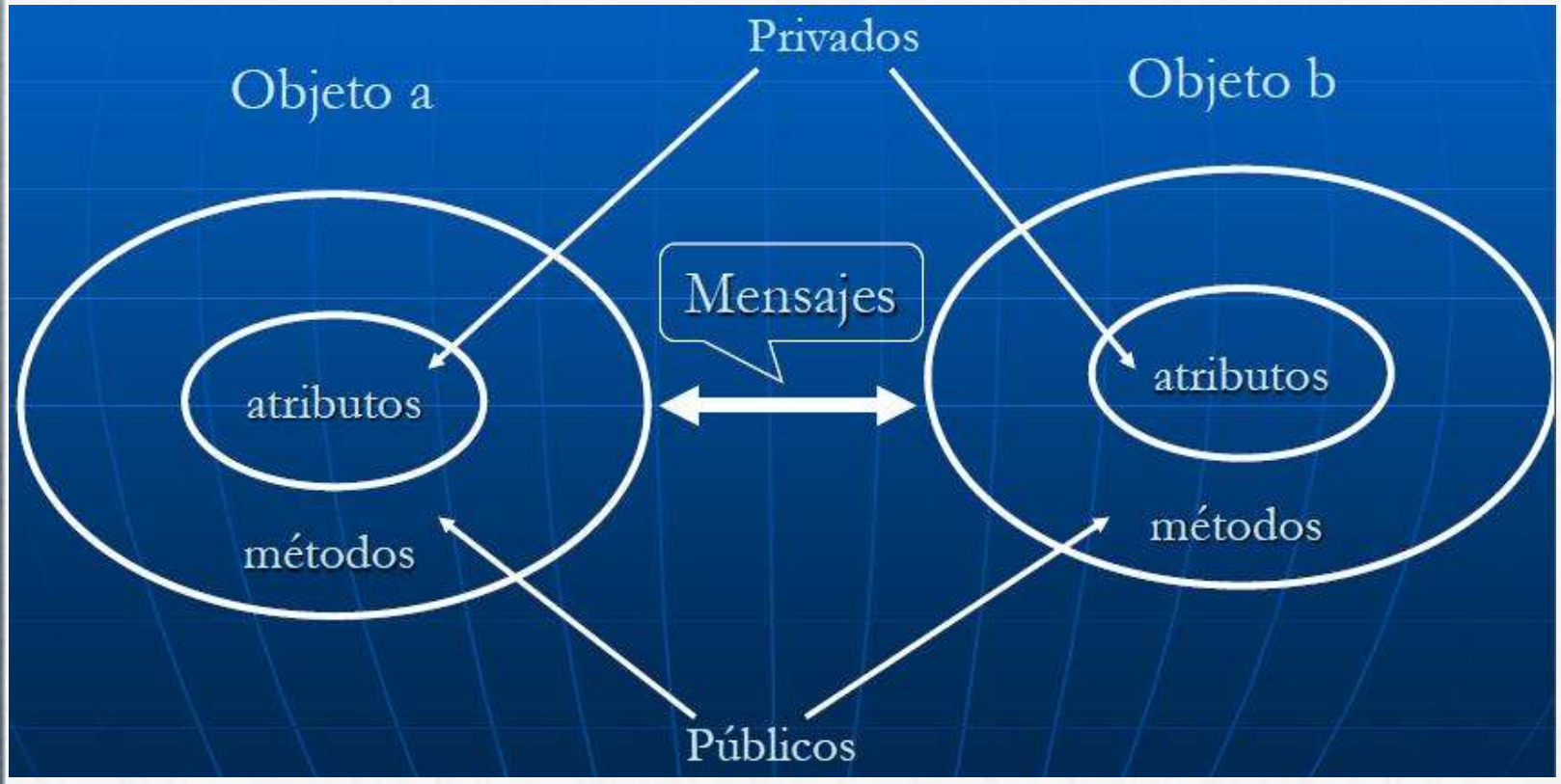


MENSAJES

- Un objeto sin comunicación con el mundo exterior no es de utilidad. La idea no es crear islas de objetos si no objetos relacionados.
- Los objetos interactúan entre ellos mediante ***mensajes.***
- El envío de un mensaje a una instancia de una clase produce la ejecución de un método.
- El paso de mensajes es el término utilizado para referirnos a la **invocación o llamada de una función miembro de un objeto.**
- ***Cuando un objeto A quiere que otro objeto B ejecute una de sus funciones o procedimientos miembro (Métodos de B), el objeto A manda un mensaje al objeto B.***

MENSAJES

- ***Cuando un objeto A quiere que otro objeto B ejecute una de sus funciones o procedimientos miembro (Métodos de B), el objeto A manda un mensaje al objeto B.***



MENSAJES

- **Ejemplo:** Una persona desea llevar su televisor descompuesto para que sea arreglado por un técnico.

persona

técnico

Entregar ( )

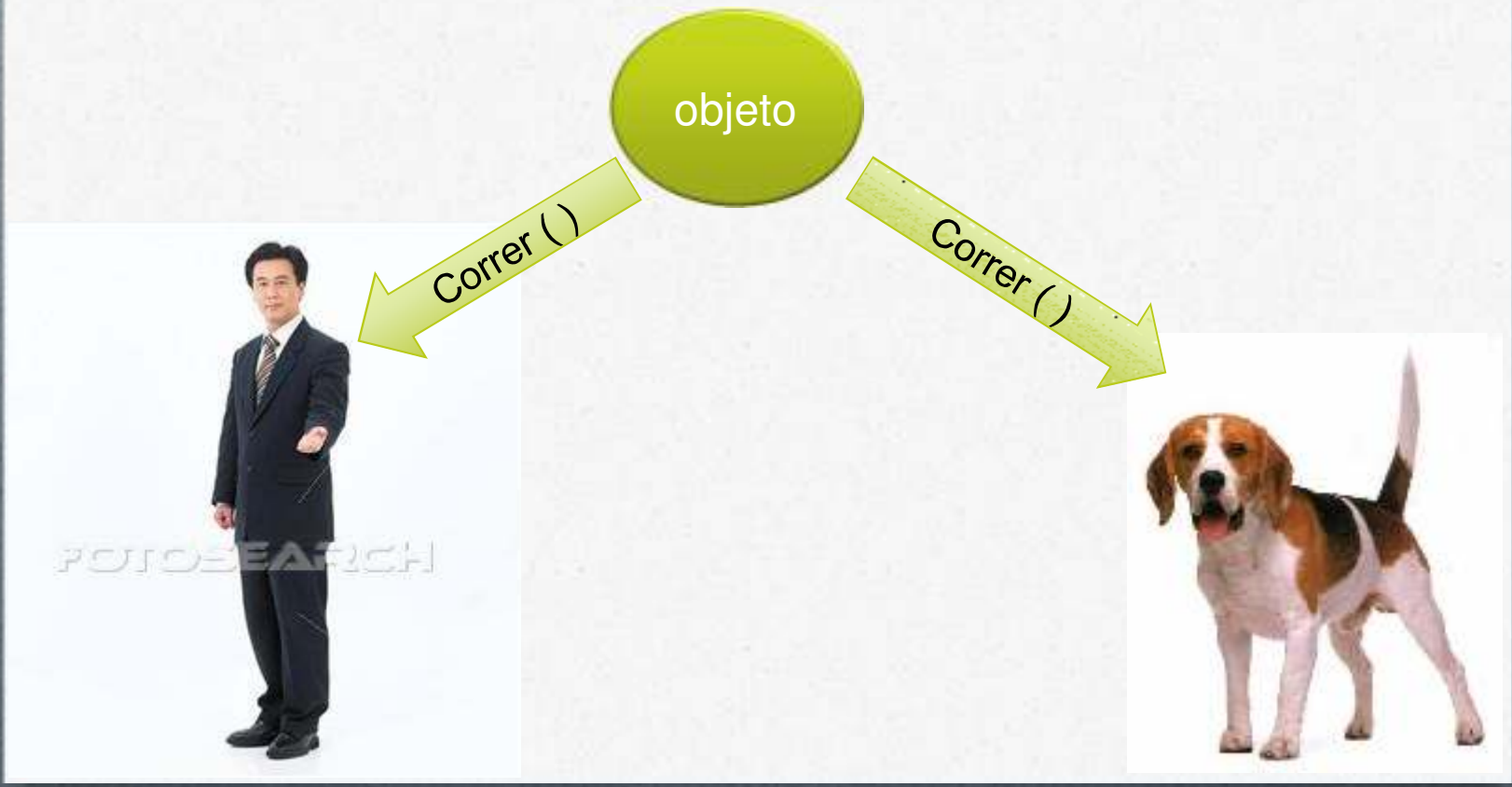
Llevar (TV)

- Ver_TV( )
- Escuchar ( )
- Apagar ( )
- Prender ( )
- Llevar (TV)

- Desatornillar ( )
- Diagnosticar ( )
- Reparar ( )
- Entregar ( ) as TV

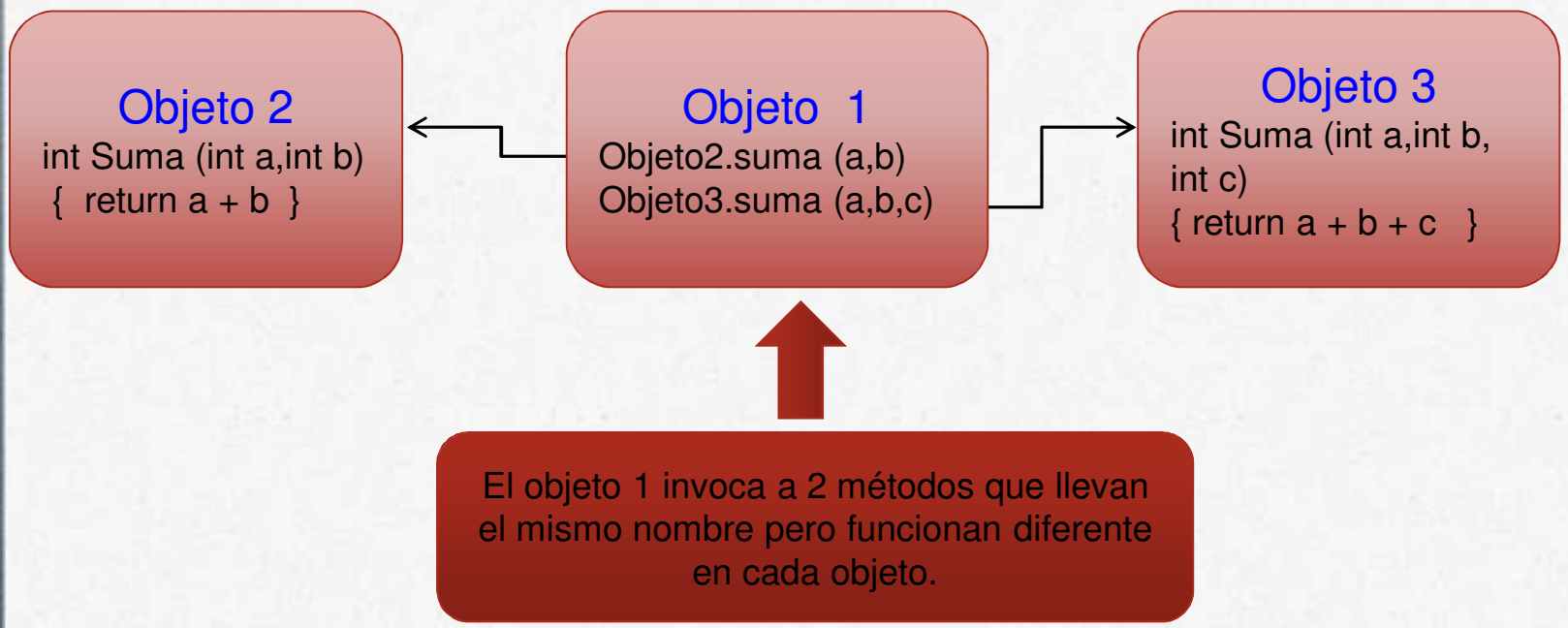
POLIMORFISMO

- Permite usar un nombre para varios propósitos relacionados, pero ligeramente diferentes.



POLIMORFISMO

- Los comportamientos pueden ser identificados bajo el mismo nombre pero procesan información de manera diferente de acuerdo al objeto que lo contenga.



MODIFICADORES DE ACCESO

- Permiten controlar la forma de acceder a los atributos y métodos encapsulados dentro de una clase.

- **TIPOS**:

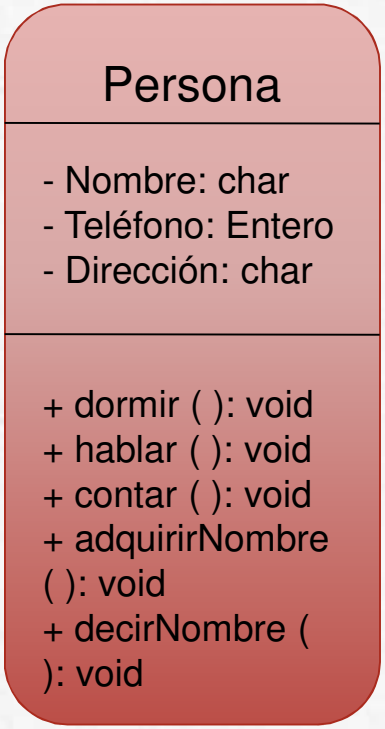
  - **PUBLICO:** Cualquier atributo o método Publico puede se accedido desde fuera de la clase. Se representa por (+).

  - **PRIVADO:** Cualquier atributo o método Privado NO puede se accedido desde fuera de la clase. Solo puede ser utilizado internamente en la clase. Se representa por (-).

  - **PROTEGIDO:** Cualquier atributo o método Protegido puede ser heredado por otra clase pero en esta ultima se convierten en elementos Privados. Se representa por (#).

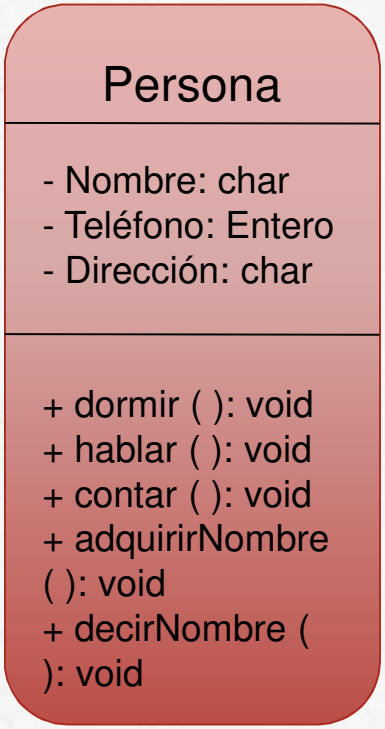
DIAGRAMA DE CLASE

- El diagrama de clase es una representación semi-gráfica de la clase que ayuda al programador a visualizar cuales son las propiedades y métodos que contendrá una clase o conjunto de clases en particular.



DECLARACIÓN DE UNA CLASE

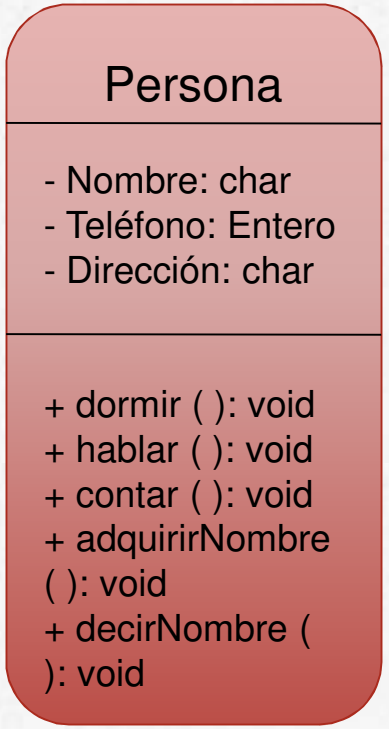
```
class Nombre_Clase
{
    public:
        //miembros públicos;
    private:
        //miembros privados;
    protected:
        // miembros protegidos;
};
```



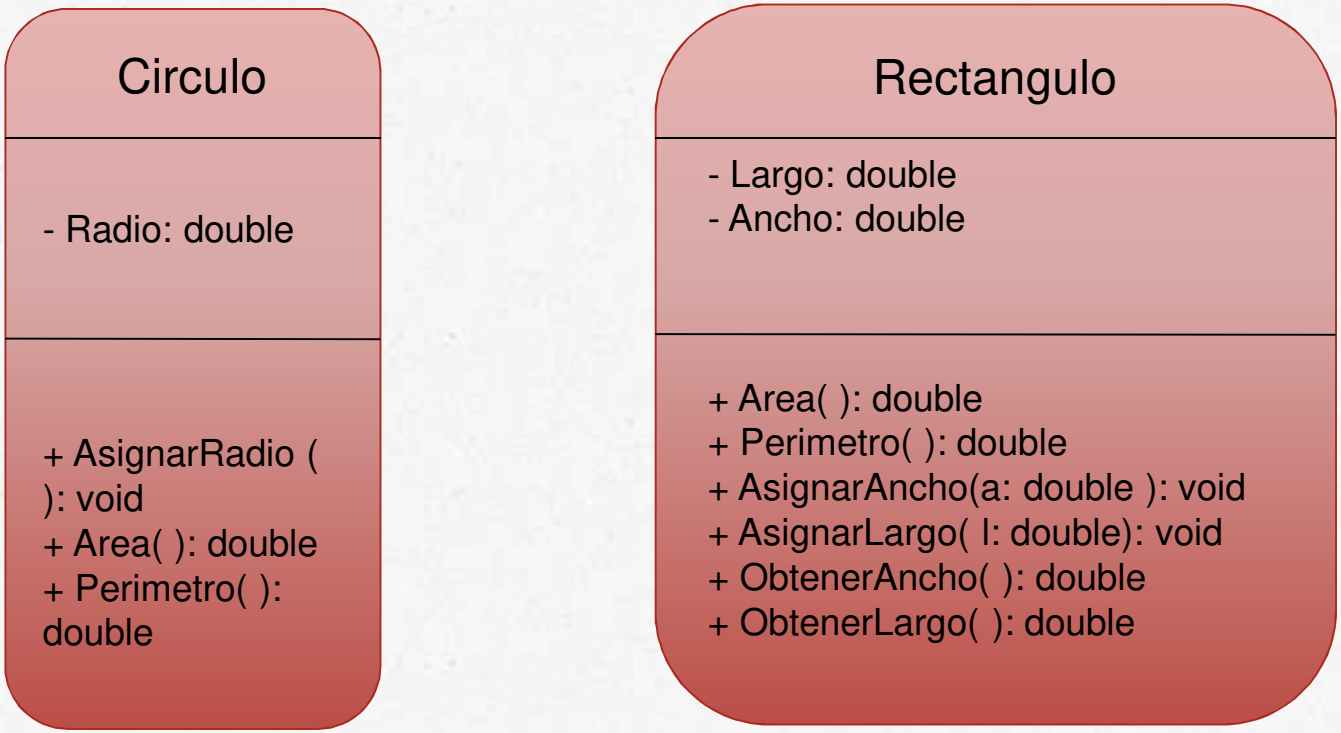
DECLARACIÓN DE UN OBJETO

Nombre_Clase Nombre_Objeto;

Nombre_Objeto.NombreMiembro;



CLASES


Circulo
- Radio: double
+ AsignarRadio ( ): void
+ Area( ): double
+ Perimetro( ): double
Rectangulo
- Largo: double
- Ancho: double
+ Area( ): double
+ Perimetro( ): double
+ AsignarAncho(a: double): void
+ AsignarLargo(l: double): void
+ ObtenerAncho( ): double
+ ObtenerLargo( ): double

CONSTRUCTORES

- Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara.
- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, por lo tanto no retornan ningún valor, ni siquiera void.
- No pueden ser heredado.
- Deben ser públicos (no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado).

CONSTRUCTORES

- Una clase puede tener cualquier cantidad de constructores, incluso ninguno. En este último caso, el compilador automáticamente crea uno para esta clase.
- El constructor predeterminado es aquel que no tiene parámetros o posee una lista de parámetros, donde todos usan argumentos predeterminados.
- Se utilizan para inicializar los objetos.
- Un constructor de copia le permite crear instancias de clase copiando los datos de instancia existentes.

CONSTRUCTORES

- La sintaxis general para los constructores es:

```
class nombreClase
{
    public:
        nombreClase( ); //Constructor predeterminado
        nombreClase(const nomClase& c); //Constructor de copia
        nombreClase(<lista de parametros>); //otro constructor
};
```

DESTRUCTORES

- Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal, por ejemplo liberando la memoria dinámica utilizada por dicho objeto o liberando recursos usados, como ficheros, dispositivos, etc.

- También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.

- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.

- No tienen parámetros.

- No pueden ser heredados.

DESTRUCTORES

- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.
- Una clase no puede tener mas de un destructor. Además, si omites el destructor, el compilador lo crea automáticamente.
- El sistema en tiempo de ejecución llama automáticamente a un destructor de la clase cuando la instancia de esa clase cae fuera de alcance o cuando la instancia se borra explícitamente.

DESTRUCTORES

- Las clases de C++ pueden contener destructores que eliminan automáticamente las instancias de la clase.

- La sintaxis general para los destructores es:

```
class nombreClase
{
    public:
        nombreClase( ); //Constructor predeterminado
        //otros constructores

        ~nombreClase( ); //destructor
        //Otros métodos miembro
};
```

SOBRECARGA DE FUNCIONES

- En C++ podemos definir varias funciones con el mismo nombre, con la única condición de que el número y/o el tipo de los argumentos sean distintos.
- El compilador decide cual de las versiones de la función usará después de analizar el número y el tipo de los parámetros. Si ninguna de las funciones se adapta a los parámetros indicados, se aplicarán las reglas implícitas de conversión de tipos.
- Las ventajas son más evidentes cuando debemos hacer las mismas operaciones con objetos de diferentes tipos o con distinto número de objetos.
- Las funciones serán ejecutadas mediante llamadas, y por lo tanto sólo habrá una copia de cada una.

SOBRECARGA DE FUNCIONES

Ejemplo:

```
int mayor(int a, int b);
char mayor(char a, char b);
double mayor(double a, double b);

int main()
 {
    cout << mayor('a', 'f') << endl;
    cout << mayor(15, 35) << endl;
    cout << mayor(10.254, 12.452) << endl;
    return 0;
}
```

SOBRECARGA DE FUNCIONES

```
int mayor(int a, int b)
{
    if(a > b) return a; else return b;
}
char mayor(char a, char b)
{
    if(a > b) return a; else return b;
}
double mayor(double a, double b)
{
    if(a > b) return a; else return b;
}
```

SOBRECARGA DE FUNCIONES

Otro ejemplo:

```
int mayor(int a, int b);
int mayor(int a, int b, int c);
int mayor(int a, int b, int c, int d);

int main()
{
    cout << mayor(10, 4) << endl;
    cout << mayor(15, 35, 23) << endl;
    cout << mayor(10, 12, 12, 18) << endl;
    return 0;
}
```

SOBRECARGA DE FUNCIONES

```
int mayor(int a, int b)
{
    if(a > b) return a; else return b;
}
int mayor(int a, int b, int c)
{
    return mayor(mayor(a, b), c);
}
int mayor(int a, int b, int c, int d)
{
    return mayor(mayor(a, b), mayor(c, d));
}
```

REFERENCIAS

- Una referencia es un nombre alternativo (un sinónimo) para un objeto o variable.
- Se puede utilizar a ambos lados del operador de asignación.
- Siempre que se declare una referencia debe ser inicializada, a menos que sea un parámetro de una función.
- Sintaxis:

  tipo& NombreReferencia = objeto;
- Ejemplo:

```
int y = 10;
int& x = y;          //x es una referencia de y
```

REFERENCIAS

```
class CReferencia
{
   private:
          int x;
   public:
          int& Valor()
          { return x;}
};
```

```
void main()
{
     CReferencia obj;
     int y;
     obj.Valor()=10;
     y=obj.Valor();
     cout<<y<<endl;
}
```

REFERENCIAS

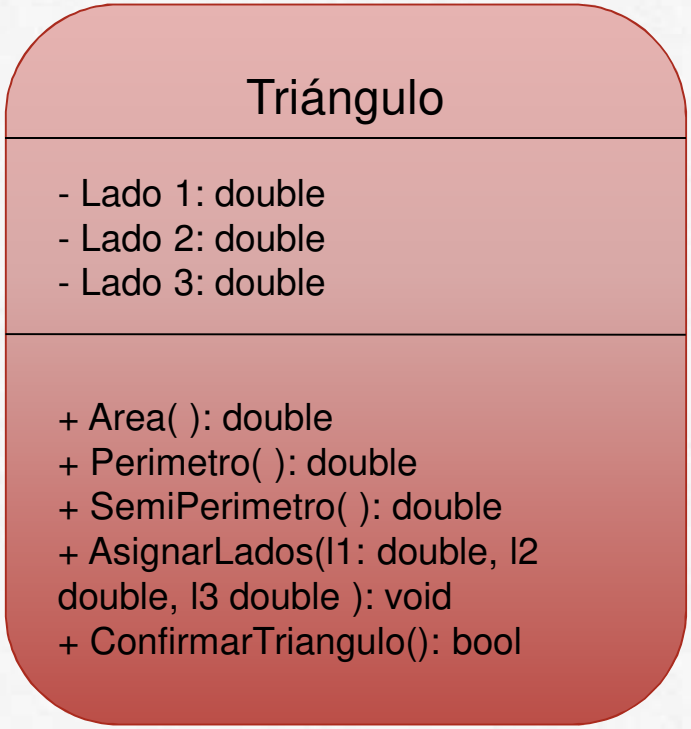
PASO DE PARÁMETROS

Se pueden pasar parámetros a funciones de 2 formas:

1. Pasar la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un apuntador.

2. Declarar el parámetro formal, como una referencia al parámetro actual que se quiere pasar como referencia.

TAREA


Triángulo
- Lado 1: double
- Lado 2: double
- Lado 3: double
+ Area( ): double
+ Perimetro( ): double
+ SemiPerimetro( ): double
+ AsignarLados(l1: double, l2
double, l3 double): void
+ ConfirmarTriangulo(): bool

SOBRECARGA DE OPERADORES

- Al igual que sucede con las funciones, en C++ los operadores también pueden sobrecargarse.
- En realidad la mayoría de los operadores en C++ ya están sobrecargados. Por ejemplo el operador + realiza distintas acciones cuando los operandos son enteros, o en coma flotante.
- En otros casos esto es más evidente, por ejemplo el operador * se puede usar como operador de multiplicación o como operador de indirección.
- C++ permite al programador sobrecargar a su vez los operadores para sus propios usos o para sus propios tipos.

SOBRECARGA DE OPERADORES

- Cuando se sobrecarga un operador, éste conserva su propiedad de binario o unario, y mantiene invariable su propiedad de evaluación y su asociatividad.
- Un operador unario se aplica sobre un solo operando (-, ~) y un operador binario sobre dos operandos (+, /).
- Los operadores sobrecargados son normalmente utilizados con clases, para facilitar determinadas operaciones con los objetos de las mismas.
- Útiles cuando se trabaja con tipos abstractos de datos que definen objetos pertenecientes al campo de las matemáticas.

SOBRECARGA DE OPERADORES

- La sobrecarga de un operador unario utilizando una función externa debe tomar un parámetro, y dos cuando se sobrecargue un operador binario.
- Cuando la función que sobrecarga un operador es un método de una clase, existe un parámetro implícito (this), se trata del objeto para el que es invocado el método.
- Por lo tanto, el método utilizado para sobrecargar un operador unario tendrá cero parámetros explícitos y un binario tendrá uno.

SOBRECARGA DE OPERADORES

- Operadores que se pueden sobrecargar:

+	**-**	*****	**/**	**%**	**^**	**&**	**|**
~	**!**	**,**	**=**	**<**	**>**	**<=**	**>=**
++	**--**	**<<**	**>>**	**==**	**!=**	**&&**	**||**
+=	**-=**	***=**	**/=**	**%=**	**^=**	**&=**	**|=**
<<=	**>>=**	**[ ]**	**( )**	**->**	**->***	**new**	**delete**

- Operadores que no se pueden sobrecargar:

.	**.***	**::**	**?:**

SOBRECARGA DE OPERADORES

- Sintaxis:

  <tipo> operator<operador binario>(<tipo> <identificador>);

- Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador, tanto en el valor de retorno como en el parámetro.

- Al sobrecargar operadores binarios, el tipo del valor de retorno y el de los parámetros no está limitado.

- Por ejemplo, si queremos sobrecargar el operador suma para complejos, tendremos que sumar dos números complejos y el resultado será un número complejo.

SOBRECARGA DE OPERADORES

- Sin embargo, C++ nos permite definir el operador suma de modo que tome un complejo y un entero y devuelva un valor en coma flotante.

```
/* Definición del operador + para complejos */
complejo operator +(complejo a, complejo b)
{ complejo temp = {a.re+b.re, a.im+b.im}; return temp; }

/* Definición del operador + para un complejo y un float */
complejo operator +(complejo a, float b)
{ complejo temp = {a.re+b, a.im}; return temp; }

/* Definición del operador + para un complejo y un entero */
int operator +(complejo a, int b) { return int(a.b)+b; }
```

SOBRECARGA DE OPERADORES

- Cuando en una clase no se define el operador de asignación, el compilador de C++ define uno por omisión.

```
/* Definición del operador + para complejos */
complejo& complejo::operator =(complejo& c)
{    real = c. real;
     imag = c.imag;
     return *this; }
```

- La referencia devuelta al objeto asignado, permite realizar asignaciones múltiples (a = b = c).

APUNTADOR THIS

- Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.
- Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.
- Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, pero si sólo hay una copia de la función y varios objetos de esa clase, ¿cómo hace la función para referirse a un dato de un objeto en concreto?

APUNTADOR THIS

- La respuesta es: usando el puntero especial llamado **this**. Se trata de un puntero que tiene asociado cada objeto y que apunta a si mismo. Ese puntero se puede usar, y de hecho se usa, para acceder a sus miembros.

Ejemplo:

```
class clase {
public:
    clase() {}
    void EresTu(clase& c) {
        if(&c == this) cout << "Sí, soy yo." << endl;
        else cout << "No, no soy yo." << endl; }
};
int main() {
clase c1, c2;        c1.EresTu(c2);        c1.EresTu(c1);
```

FUNCIONES AMIGAS

- La utilidad de las funciones amigas es poder acceder a los datos privados de una o más clases.
- Una función declarada friend de una clase C, es una función no miembro de la clase, que puede acceder a los miembros privados de la clase.
- Una función amiga puede declararse en cualquier sección de la clase.
- No es miembro de la clase, por lo que no se ve afectada por los modificadores protected, private y public.
- Una función puede ser amiga de una clase y miembro de otra.

FUNCIONES AMIGAS

- La implementación de las funciones amigas no hacen uso del operador de ámbito (::) porque es una función amiga de la clase, pero no pertenece a la clase.
- La llamada no necesita hacerse a través de un objeto de la clase.
- Las funciones amigas no contienen el argumento implícito *this*.
- Es la clase la que dice quiénes son sus amigos y pueden acceder a sus miembros privados.
- Ninguna función puede autodeclararse amiga y acceder a la privacidad de una clase sin que la propia clase tenga conocimiento de ello.

FUNCIONES AMIGAS

Ejemplo:

```
class Punto {
 private:
    float x, y;
 public:
    Punto (float a, float b);
    void visualizar ();
    float distancia (Punto p);
 };

float Punto:: distancia ( Punto p) {
  float d;
  d = sqrt( sqrt(p.x-x)+sqrt(p.y - y) );
  return d; }
```

FUNCIONES AMIGAS

- De esta forma el llamado a la función es poco elegante:

```
int main() {
   Punto c1(5, 2), c2(2,3);
   dist = c1.distancia(c2);
}
```

- El método distancia tiene acceso a los atributo del objeto c1 como a los atributos del objeto c2. Sería más vistoso llamar la función de esta forma:

```
dist = distancia (c1, c2);
```

- Para ello se requiere que distancia se una función de la forma:

```
float distancia (Punto p1, Punto p2);
```

FUNCIONES AMIGAS

- Dentro de la clase quedaría como:

  friend float distancia (Punto p1, Punto p2);

- Y su definición:

```
float distancia ( Punto p1, Punto p2)
{ float d;
  d= sqrt( sqrt(p2.x-p1.x)+sqrt(p2.y - p1.y) );
  return d;
}
```

CLASES AMIGAS

- Cuando se requiere que todos los métodos de una clase C1 sean amigos de otra clase C2, hay que declarar la clase C1 amiga de la clase C2.

```
class C1
{
    //miembros
};
class C2
{
    friend class C1;
    // miembros
};
```

CLASES AMIGAS

El modificador **friend** puede aplicarse a funciones o a toda una clase para inhibir el sistema de protección.

Las relaciones de "amistad" entre clases son parecidas a las amistades entre personas:

- La amistad no puede transferirse, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C. (La famosa frase: "los amigos de mis amigos son mis amigos" es falsa en C++, y probablemente también en la vida real).

- La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C. (Los hijos de mis amigos, no tienen por qué ser amigos míos. De nuevo, el símil es casi perfecto).

CLASES AMIGAS

- La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A. (En la vida real, una situación como esta hará peligrar la amistad de A con B, pero de nuevo me temo que en realidad se trata de una situación muy frecuente, y normalmente A no sabe que B no se considera su amigo).

Ejemplo:
```
class Camion;   //referencia anticipada
class Coche {
private:
    int plazas, velocidad;
public:
    Coche (int p, int v);
    void masVeloz (Camion t);
};
```

CLASES AMIGAS

```
void Coche::masVeloz ( Camion t) {
{
        int comp;
        comp = velocidad - t.velocidad;
        if (comp < 0)
                cout << " El camión es más rápido " ;
        else if (comp == 0)
                cout << " Son igual de rápidos " ;
        else
                cout << " El coche es más rápido " ;
}
```

CLASES AMIGAS

- El método masVeloz accede al atributo velocidad de la clase Camion, sin embargo esto no esta permitido a menos que la clase Camion lo permita.

Ejemplo:
```
class Camion{
    private:
        int peso, velocidad;
    public:
        Camion (int p, int v);
        friend class Coche;
        friend void Coche::masVeloz (Camion t);
};
```

- Al declarar el método masVeloz de la clase Coche, o a la clase Coche completa como amiga de la clase Camion, ésta le permitirá tener acceso a sus miembros privados.

HERENCIA

- Una de las principales propiedades de las clases es la *herencia*. Esta propiedad nos permite crear nuevas clases a partir de clases existentes, conservando las propiedades de la clase original y añadiendo otras nuevas.

- Esta nueva clase se denomina subclase o clase derivada y la clase a partir de la que se deriva se denomina clase existente, superclase o clase base.

- La nueva clase definida a partir de la clase existente, adopta todos los miembros de la clase existente:
  - - atributos
  - - métodos

HERENCIA

- En las funciones miembro definidas en la subclase, no se tiene acceso a lo privado de la superclase, a pesar de ser miembros que se heredan.
- El proceso de herencia no afecta de ningún modo a la clase base.
- La clase derivada hereda todas las características de la clase base.
- La clase derivada puede definir características adicionales.
- La clase derivada puede redefinir características heredadas de la clase base.
- La clase derivada puede anular características heredadas de la clase base.

HERENCIA

- Sintaxis:

```
class Subclase : public/protected/private Superclase
{
    // lista de miembros públicos y privados
};
```

HERENCIA

Ejemplo:

```
class Mamifero
{
    int edad;
    float peso;
public:
    Mamifero (int e = 0, float p = 0) : edad(e), peso (p) { }
    void SetEdad (int e);
    void SetPeso (float p);
    int GetEdad ();
    float GetPeso ();
};
```

HERENCIA

- Un **método virtual** es un miembro de una clase que puede ser redefinido en cada una de las clases derivadas de ésta, y una vez redefinido puede ser accedido mediante un apuntador o una referencia a la clase base.

- La redefinición de un método virtual en una clase derivada debe tener el mismo nombre, número y tipos de parámetros, y tipo de valor retornado que en la clase base.

- Un método se declara virtual escribiendo la palabra clave **virtual** al principio de la declaración del método en la clase donde aparece por primera vez.

HERENCIA

```
class A
{
    virtual void fun();
}
```

- Para acceder a un atributo o método que ha sido redefinido en la clase derivada, se utiliza el nombre de su clase más el operador de ámbito **::**.

HERENCIA

```
class A;                    //clase base
class B: private A;         //clase derivada privada
class C: protected A;       //clase derivada protegida
class D: public A;          //clase derivada pública
```

- **privada**: sus miembros public y protected pasan a ser private en la clase derivada y solo serán accesibles por los métodos miembros o amigos de la derivada.
- **protegida**: sus miembros public y private pasan a ser protected en la clase derivada y solo serán accesibles por los métodos miembros o amigos de la derivada.
- **pública**: sus miembros públicos, privados y protegidos siguen siendo públicos, privados y protegidos en la clase derivada respectivamente.

SEGURIDAD DE CLASES

Accedido desde:	Un miembro que es:		
	Privado	**Protegido**	**Público**
Su misma clase (métodos y funciones amigas)	Sí	Sí	Sí
Cualquier clase derivada	No	Sí	Sí
Cualquier clase no derivada (y no amiga)	No	No	Sí
Cualquier función externa	No	No	Sí

HERENCIA

- Una clase derivada no tiene acceso directo a los miembros privados de su clase base, pero si puede acceder a los miembros públicos y protegidos.
- Los miembros protegidos de la clase base pasan a ser miembros privados de la clase derivada, y por lo tanto ésta tendrá acceso directo a ellos.
- Una clase derivada puede añadir sus propios atributos y métodos. Si el nombre de alguno de estos coincide con el de un miembro heredado, este último queda oculto para la clase derivada.
- Los miembros heredados por una clase derivada pueden, a su vez, ser heredados por más clases derivadas de ella.

HERENCIA

En resumen …

Los miembros de una clase pueden ser:

- **private**:
  - Miembros privados. Protegidos de todo acceso fuera del ámbito de la clase.
- **protected**:
  - Miembros protegidos. Protegidos de todo acceso fuera del ámbito de su clase o de sus clases derivadas.
- **public**:
  - Miembros públicos. Accesibles desde cualquier ámbito.

HERENCIA

La clase derivada hereda todos los atributos y todos los métodos, excepto:

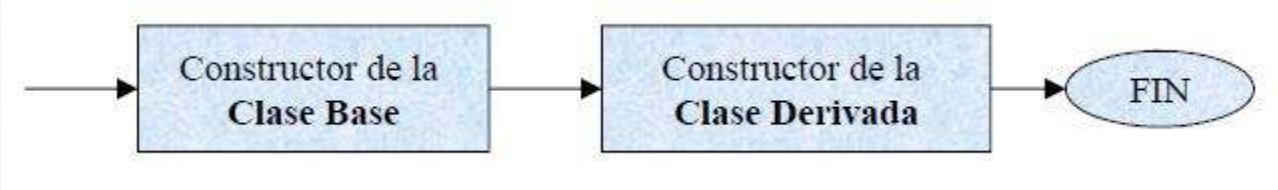
- **Constructores y destructor**: Si en la subclase no están definidos, se crea un constructor por defecto y un destructor, aunque sin que hagan nada en concreto.
- **Operador de asignación**: Si en la subclase no está definido, se crea uno por defecto que se basa en el operador de asignación de la superclase.

Se deben crear los constructores y el destructor en la subclase.

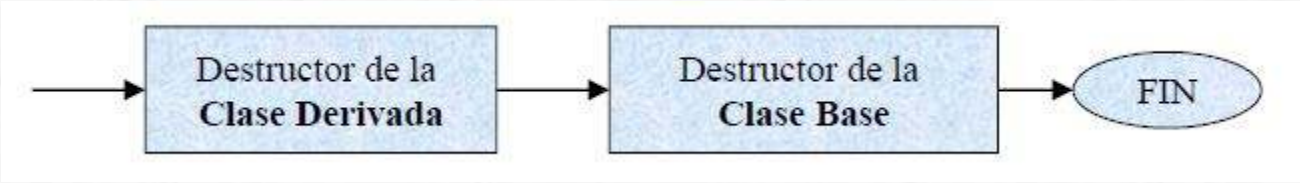
Se debe definir el operador de asignación en la subclase.

HERENCIA

- Cuando se crea un objeto de una clase derivada, se invoca a su constructor, que a su vez invoca al constructor sin parámetros de la clase base, que su vez invoca al constructor de su clase base, y así sucesivamente. Por lo tanto; primero se ejecutan los constructores de las clases base de arriba abajo en la jerarquía de clases y finalmente el de la clase derivada.



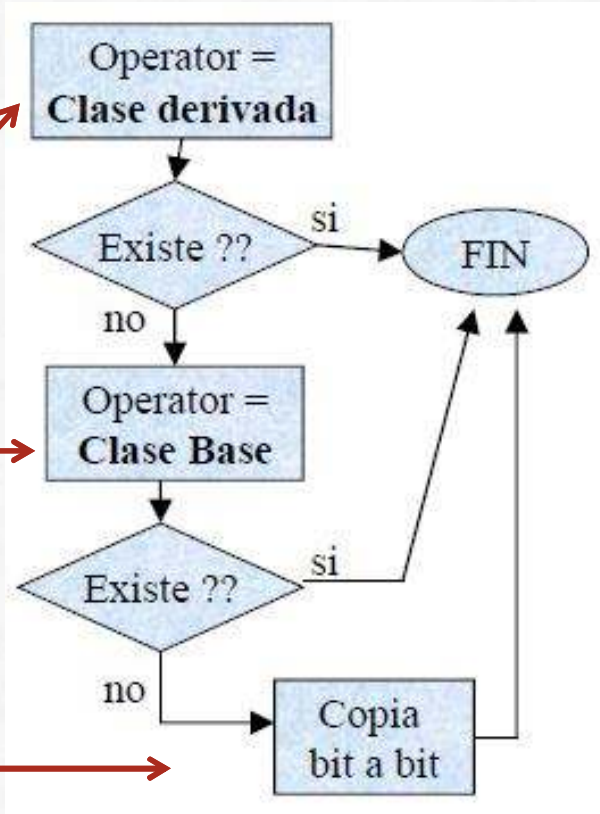
- Los destructores se ejecutan en orden inverso.



HERENCIA

El operador de asignación (=) se comporta de la siguiente manera:

- Se invoca el operador de asignación de la clase derivada.
- Se invoca al operador de asignación de la superclase, que afecta a los atributos heredados. El resto de atributos se copian bit a bit.
- Si no existe ninguno de los dos, la copia se realiza bit a bit.



Funciones Inline

- Son aquellas definidas dentro de la declaración de la clase.
- Cuando el programa se compila, el código generado para una función inline se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada.
- Esto nos proporciona una ventaja, el código de estas funciones se ejecuta más rápidamente, ya que se evita usar la pila para pasar parámetros y se evitan las instrucciones de salto y retorno.
- También tiene un inconveniente: se generará el código de la función tantas veces como ésta se use, con lo que el programa ejecutable final puede ser mucho más grande.

Funciones Inline

- Por lo anterior, se recomienda declarar sólo funciones pequeñas como inline.

- También es posible definir una función fuera de la declaración de clase (en el archivo NombreClase.cpp). En este caso sólo se requiere hacer saber al compilador de manera explícita que la función será inline mediante el uso del modificador **inline** al inicio de su cabecera:

  inline <tipo> NombreClase::NombreFuncion (<tipo> parametro) {}

Modificador const

- El modificador **const**, utilizado para una función, proporciona ciertos mecanismos necesarios para mantener la protección de los datos.
- Cuando una función miembro no deba modificar el valor de ningún dato de la clase, se puede y se debe declarar como constante. Esto evitará que la función intente modificar los datos del objeto.
- Al declarar una función como const, el compilador generará un error durante la compilación si la función intenta modificar alguno de los datos miembro del objeto, ya sea directamente o de forma indirecta.
- Si la función const invoca a otras funciones, usar este modificador asegura que ni siquiera esas funciones puedan modificar los datos del objeto.

Modificador const

- Puede ocurrir que otros programadores pueden usar clases definidas por uno, o bien, uno puede usar clases definidas por otros programadores. En ese caso es frecuente que sólo se disponga de la declaración de la clase, y el modificador "const" nos dice si cierto método modifica o no los datos del objeto.

Modificador static

- Ciertos miembros de una clase pueden ser declarados como **static**. Los miembros **static** tienen algunas propiedades especiales.

- En el caso de los datos miembro **static** sólo existirá una copia que compartirán todos los objetos de la misma clase. Si consultamos el valor de ese dato desde cualquier objeto de esa clase obtendremos siempre el mismo resultado, y si lo modificamos, lo modificaremos para todos los objetos.

- Los miembros **static** deben existir aunque no exista ningún objeto de la clase, y declarar la clase no crea los datos miembro estáticos, por lo que es necesario declararlos explícitamente.

Modificador static

- Al declararlos es necesario inicializarlos, pues de no hacerlo, al declarar objetos de esa clase los valores de los miembros estáticos estarían indefinidos, y los resultados no serían los esperados.

- Las funciones miembro declaradas como **static** no pueden acceder a los miembros de los objetos, sólo pueden acceder a los datos miembro de la clase que sean **static**. Esto significa que no tienen acceso al puntero **this**, y además suelen ser usadas con su nombre completo, incluyendo el nombre de la clase y el operador de ámbito (::).

  static <tipo> NombreClase::NombreFuncion (<tipo> parametro) {}