FACULTAD DE INFORMÁTICA

Introducción a la POO
Clases y objetos

TEMA

Programación orientada a objetos — Unidad 1

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Contenido

FdI UCM

Orientado a objetos

¿Qué se puede calificar como *orientado a objetos*?

El término “orientado a objetos” ...

... ¡está de moda! ... ¡vende!

Uso inadecuado por intereses comerciales.

Los sistemas de *manipulación de objetos* (por ejemplo, interfaces gráficas o programas de dibujo que manejan objetos gráficos) no son necesariamente sistemas *orientados a objetos*.

La orientación a objetos se refiere a una metodología de desarrollo de software.

- ✓ Una nueva forma de enfocar el diseño y la estructura de los programas (ingeniería del software), así como de escribirlos (programación).
- ✓ Técnicas y mecanismos que facilitan el desarrollo orientado a objetos de las aplicaciones.

Un poco de historia

Los objetos llevan tiempo en el mundo de la programación.

El concepto de objeto aparece a finales de los años 60 en el lenguaje Simula-67 (desarrollado en Noruega).

El lenguaje Simula influye en el trabajo del Software Concepts Group (XEROX PARC, centro de investigación en Palo Alto, California):

- ✓ Proyecto Dynabook del visionario Alan Kay.
  Primer proyecto de computadora personal de uso general.
- ✓ PC pequeño, barato y con buenas prestaciones:
  - Interacción por medio de un nuevo dispositivo: el ratón.
  - Interfaz gráfica (ventanas, menús, botones, ...)
  - Sencillo manejo.
- ✓ Entorno ampliable por el usuario mediante un sencillo lenguaje de programación: Smalltalk.
- ✓ Versión final del lenguaje: Smalltalk-80 (actual: VisualWorks).

Un poco de historia

Smalltalk, lenguaje de programación orientada a objetos puro, no gozó de popularidad, pero su entorno influyó notablemente otros desarrollos (Macintosh).

En los años 80 renace el interés por la orientación a objetos:

- ✓ Conferencias internacionales específicas (OOPSLA).
- ✓ Extensión de las técnicas a la Ingeniería del software.
- ✓ Desarrollo de lenguajes de programación orientados a objetos (Eiffel) e incorporación de estas tecnologías en lenguajes existentes (C++, Turbo Pascal, ...)

En los años 90 los objetos se hacen omnipresentes:

- ✓ Sistemas operativos orientados a objetos.
- ✓ Entornos de desarrollo orientado a objetos con muchas facilidades para la programación (método visual).
- ✓ SUN desarrolla el lenguaje Java.

Evolución de los enfoques de programación

A lo largo del tiempo las técnicas de programación se han ido centrando en distintas entidades de programa, cada vez con un mayor nivel de abstracción:

Los procedimientos (subrutinas en general)
⇒ La programación procedimental.

Los módulos
⇒ La programación modular.

Los tipos abstractos de datos (TAD)
⇒ La programación basada en tipos.

Las clases de objetos
⇒ La programación orientada a objetos.

Cada enfoque precisa de determinados mecanismos, aquellos que permiten implementar las entidades en que se basan.

Evolución de los enfoques de programación

Programación procedimental

Decidir qué procedimientos se quieren;
utilizar los mejores algoritmos que se pueda.

Diseño conducido por el procesamiento (procedimientos que implementan algoritmos que realizan tareas concretas).

Mecanismos:

- ✓ Procedimientos (funciones, rutinas, ...)
- ✓ Paso de argumentos.

Lenguajes: Fortran, C, Pascal.

```
double sqrt(double arg)
{
   // Cálculos
}
```

```
void func()
{
   double raiz2 = sqrt(2);
   // ...
}
```

Evolución de los enfoques de programación

Programación modular

Decidir qué módulos se quieren; partir el programa de forma que los datos estén ocultos en los módulos.

Diseño centrado en la organización de los datos.

Los módulos como conjuntos de procedimientos relacionados junto con los datos que manipulan.

Mecanismos:

- ✓ Los de la programación procedimental.
- ✓ Ocultamiento de los datos.
- ✓ Interfaz que determina la forma de utilización.
- ✓ Implementación variable.
- ✓ Compilación separada.

Lenguajes: C, Modula-2.

Evolución de los enfoques de programación

Programación modular: una pila de enteros en C++

```
// PILA.H - Interfaz del módulo
int pop();
void push(int);
const max = 100;

// PILA.CPP - Implementación del módulo
#include <PILA.H>
static int v[max]; // 'static' hace que sean locales
static int* p = v; // a este módulo (ocultamiento)
       // pila inicializada a vacía

int pop()
{ ... }

void push(int i)
{ ... }
```

Interfaz que permanece

Implementación intercambiable

¿Qué ocurre si necesitamos dos o más pilas?

Evolución de los enfoques de programación

Abstracción de datos (*tipos abstractos de datos)*

Decidir qué tipos se quieren; proporcionar un conjunto completo de operaciones para cada tipo.

Diseño centrado en los tipos de datos.

Módulos que implementan distintos tipos de datos.

Tipo + Operaciones definidas sobre los valores de ese tipo.

Mecanismos:

- ✓ Los de la programación modular.
- ✓ Encapsulación de código y datos.

Lenguajes: Modula-2, Ada, C++.

Evolución de los enfoques de programación

Abstracción de datos: un tipo **Complejo** en C++

```
// Encapsulación de código y datos en una estructura
class Complejo {
public:
  Complejo(double r, double i) { real = r; imag = i; }
  Complejo(double r) { real = r; imag = 0; }

  friend Complejo operator+(Complejo, Complejo);
  friend Complejo operator-(Complejo, Complejo);
  friend Complejo operator-(Complejo); // menos unario
  friend Complejo operator*(Complejo, Complejo);
  friend Complejo operator/(Complejo, Complejo);

  // ...
private:
  double real, imag; // Privado: inaccesible
};
```

INTERFAZ

Evolución de los enfoques de programación

Abstracción de datos: un tipo **`Complejo`** en C++

Datos protegidos: accesibles únicamente en las operaciones.

Operaciones (funciones) implementadas aparte:

```
// Suma de complejos
Complejo operator+ (Complejo c1, Complejo c2)
{
  return Complejo(c1.real + c2.real, c1.imag + c2.imag);
}
```

Uso similar al del resto de los tipos de datos.

```
Complejo a = 2.3;
Complejo b = 1 / a;
Complejo c = a + b * Complejo(1, 2.3);
// ...
c = -(a / b) + c;
```


Evolución de los enfoques de programación

Problemas con la abstracción de datos

La adaptación o especialización no se maneja adecuadamente.

Por ejemplo, consideremos un programa de dibujo que contemple círculos, triángulos y cuadrados.

Asumamos que ya tenemos algunos tipos:

```
class Punto { /* ... */ };
class Color { /* ... */ };
```


Evolución de los enfoques de programación

Problemas con la abstracción de datos

Diferenciación entre las distintas formas:

```
enum Tipo { circulo, triangulo, cuadrado };
class Forma {
public:
  Punto donde() { return centro; }
  void mover(Punto hacia)
    { centro = hacia; dibujar(); }
  void dibujar();
  void rotar(int);
  // Más operaciones
private:
  Punto centro;
  Color col;
  Tipo t;
  // ...
};
```


Evolución de los enfoques de programación

Problemas con la abstracción de datos

Implementación de la función **`dibujar()`**:

```
void Forma::dibujar()
{
  switch (t) {
  case circulo:
    // dibujar el círculo
    break;
  case triangulo:
    // dibujar el triángulo
    break;
  case cuadrado:
    // dibujar el cuadrado
  }
}
```

Mala organización.

La incorporación de nuevas formas hace necesario adaptar el código de múltiples funciones.

Aumenta el riesgo de introducir errores.

Evolución de los enfoques de programación

Programación orientada a objetos

El problema del ejemplo anterior radica en que no se hace distinción entre las propiedades de cualquier forma (comunes) y las propiedades particulares de cada tipo de forma. Esa distinción es en la que se basa el enfoque orientado a objetos.

Decidir qué clases se quieren;
proporcionar un conjunto completo de operaciones en cada clase;
hacer explícitas las semejanzas mediante la herencia.

Diseño centrado en las clases de objetos.

Mecanismos:

- ✓ Los de la abstracción de datos.
- ✓ Herencia.
- ✓ Vinculación dinámica y polimorfismo.

Lenguajes: Smalltalk, C++, Eiffel.

Evolución de los enfoques de programación

Programación orientada a objetos

Lo común de las distintas formas se expresa en una superclase:

```
class Forma {
public:
  Punto donde() { return _centro; }
  void mover(Punto hacia)
    { _centro = hacia; dibujar(); }
  virtual void dibujar();
  virtual void rotar();
  // Más operaciones
private:
  Punto _centro;
  Color _col;
  // ...
};
```

virtual

Específico para formas concretas.

Implementado en las subclases.

Evolución de los enfoques de programación

Programación orientada a objetos

La superclase establece una interfaz común para las distintas formas específicas (subclases), una forma de manipulación que es común para todos los distintos tipos de formas:

```
void rotar_todas(Forma* v, int num, int angulo)
{ // rotar todas las formas que hay en el vector v
 for(int i = 0; i < num; i++) v[i].rotar(angulo);
 for(i = 0; i < num; i++) v[i].dibujar();
} // Todas las formas aceptan las operaciones rotar y dibujar
```

La especialización (subtipos) se logra con las subclases de **`Forma`**:

```
class Circulo : public Forma {
public:
  void dibujar() { /* ... */ };
  void rotar(int) {} // si, una función nula (¿rotar círculo?)
private:
  int _radio;
};
```


Lenguajes de programación orientados a objetos

En principio, cualquier técnica de programación se puede implementar con cualquier lenguaje de programación (!incluso con el lenguaje ensamblador!)

Sin embargo, los lenguajes se han diseñado para determinadas técnicas de programación, incorporando mecanismos que permiten implementar con facilidad esas técnicas.

Un lenguaje de programación se podrá considerar orientado a objetos sólo si incorpora los mecanismos necesarios para implementar en los programas las técnicas y métodos establecidos por la programación orientada a objetos:

Encapsulación, ocultamiento de información, herencia, vinculación dinámica, polimorfismo, ...

C++ es un auténtico lenguaje orientado a objetos.

Programación con tipos abstractos de datos (TAD)

Cada módulo contiene la definición (interfaz) y la implementación de un *tipo*.

El tipo se define mediante un nombre de tipo y un conjunto de operaciones que se pueden aplicar a los ejemplares (variables) de ese tipo.

La *forma* de los ejemplares (datos que contienen) se oculta, de forma que los ejemplares se manejen exclusivamente a través de las operaciones definidas.

Podemos distinguir dos vistas distintas de un TAD:

- ✓ *Vista externa*: la que se ofrece a los programadores que usan el TAD (nombre del tipo y forma de las operaciones)
- ✓ *Vista interna*: la del constructor del TAD (datos que conforman el tipo e implementación de las operaciones)

Programación con tipos abstractos de datos (TAD)

Por ejemplo, supongamos un TAD para matrices cuadradas de enteros de 10 x 10.

- ✓ Interfaz (vista externa):
  Nombre del tipo: **`Matriz`**
  Operaciones:
  +: **`Matriz`** x **`Matriz`** → **`Matriz`**
  – : **`Matriz`** x **`Matriz`** → **`Matriz`**
  `Invertir`: **`Matriz`** → **`Matriz`**
  . . .
- ✓ Vista interna (implementación):
  Datos: array bidimensional de N x N elementos enteros.
  Código de todas y cada una de las operaciones.

Programación con tipos abstractos de datos (TAD)

Interfaz (vista externa)

```
class Matriz {
public:
  friend Matriz operator+(Matriz, Matriz);
  friend Matriz operator-(Matriz, Matriz);
  friend Matriz Invertir(Matriz);
  // ...
```

Vista interna (implementación)

```
private:
  int matriz[10][10];
};

Matriz operator+(Matriz A, Matriz B)
{
  // ...
}
Matriz operator-(Matriz A, Matriz B)
{
  // ...
```


Programación con tipos abstractos de datos (TAD)

Cualquier programador que desee hacer uso del TAD encuentra todo lo que necesita en la interfaz (vista que se proporciona hacia el exterior, a los usuarios).

- ✓ Nombre del tipo: **`Matriz`**
- ✓ Forma de uso de cada operación definida: +, -, Invertir, ...

Todo lo que tiene que hacer es crear ejemplares del TAD (variables):

```
Matriz M1, M2, M3;
```

Y aplicar las operaciones según proceda:

```
M1 = M2 + M3;
M2 = Invertir(M1);
...
```

Programación con tipos abstractos de datos (TAD)

El módulo del TAD lo que contiene es una definición (nombre del tipo y forma de las operaciones establecidas).

En tiempo de ejecución, hasta que no se crean ejemplares del TAD (las variables) no existe nada relacionado con el TAD.

El TAD, al definir la forma de las operaciones, establece el comportamiento de los ejemplares o variables del tipo.

Todos los ejemplares tienen un mismo comportamiento (se pueden aplicar las mismas operaciones a todos ellos).

Cada ejemplar contiene sus propios datos (tal como se definen en el TAD), datos separados de los de los demás ejemplares.

Los valores que contienen los datos de cada ejemplar determinan el estado del ejemplar.

Cada ejemplar (variable) tiene su propio estado,
pero todos los ejemplares tienen el mismo comportamiento.

Programación con tipos abstractos de datos (TAD)

```
Matriz M1, M2, M3;
```

Las tres variables (ejemplares) **M1**, **M2** y **M3** tienen el mismo comportamiento, ya que a cualquiera de ellas se le puede aplicar la operación suma:

```
M1 + ...        M2 + ...        M3 + ...
```

y se puede invertir cualquiera de ellas:

```
Invertir(M1)      Invertir(M2)      Invertir(M3)
```

Pero cada una de ellas contendrá sus propios valores en sus datos (array bidimensional en este caso):

```
M1                M2                M3
2 4 3 ...         1 1 2 ...         6 5 4 ...
1 2 3 ...         4 4 4 ...         3 6 2 ...
...               ...               ...
```


Programación con tipos abstractos de datos (TAD)

Resumiendo:

Un TAD se identifica por su nombre y define un conjunto de operaciones que se pueden aplicar a los ejemplares del tipo.

Los ejemplares de un TAD son las variables de ese tipo.

El TAD encapsula los datos que conforman los ejemplares del tipo y el código de las operaciones establecidas.

- ✓ Los datos y la implementación de las operaciones se ocultan.
- ✓ A los utilizadores del tipo se les proporciona una visión externa constituida por la interfaz (nombre del tipo y forma de uso de las operaciones).

Todos los ejemplares tienen el mismo comportamiento.

Cada ejemplar tiene su propio estado (datos propios).

Se puede cambiar la implementación sin que varíe la interfaz. Bastará recompilar cualquier programa que use el TAD.

Programación orientada a objetos (POO)

Una nueva forma de ver las cosas.

Se puede decir que la POO es la programación con TAD más una serie de mecanismos adicionales (herencia, ...)

Dejando aparte esos otros mecanismos, podemos establecer un paralelismo entre los TAD y la POO:

	Programación con TAD	POO
Definición	TAD	Clase
Estado	Datos	Atributos
Comportamiento	Operaciones	Métodos
Ejemplares	Variables	Objetos

Clases de objetos

Una clase es una implementación de un tipo abstracto de datos.

datos	+	operaciones	TAD
atributos	+	métodos	Clase

Las clases, como los TAD, son declaraciones.

Ejemplares (lo que existe durante la ejecución):
variables para un TAD, objetos para una clase.

Los datos se identifican como **atributos** en la clase:
cada ejemplar (objeto) de la clase tendrá sus propios atributos.

Los atributos se ocultan (*ocultamiento de la información*).

Todos los objetos de una clase tienen definido un mismo conjunto de **métodos** (operaciones).

Encapsulación de código y datos: la clase como módulo.

Ocultación de información: público/privado (interfaz/implementación).

Las clases en C++: estructura `class`

Sintácticamente similar a una estructura (`struct`).

```
class Clase {
public:
  void dato(int d)
  { _dato = d; }
  int dato()
  { return _dato; }
  void mostrar();
private:
  int _dato;
};
```

Prototipos o funciones insertadas

Funciones miembro (**Métodos**)

Parte pública (servicios)

Atributo

Parte privada

Termina en ;

Los elementos son privados por defecto

Las clases en C++

Atributos y métodos

Los atributos deben estar ocultos (ser privados).
(aunque C++ permite declarar atributos públicos,
lo correcto es que los atributos sean privados).

Los métodos se implementan con funciones miembro.

La forma de los servicios (cabeceras de los métodos públicos) debe ser visible desde el exterior (pública); constituyen la interfaz.

Puede haber métodos privados (auxiliares para la clase, no identificados como servicios que ofrecen sus objetos).

Interfaz (lo público): servicios (métodos públicos).

Implementación (lo privado):
Atributos, declaración de métodos privados y código de todos los métodos (públicos y privados).

Las clases en C++

Funciones miembro insertadas

La implementación de los métodos puede estar en la propia estructura `class` en forma de *funciones miembro insertadas*.

Métodos **`dato()`** de la clase anterior:

```
void dato(int d) { _dato = d; }
int dato() { return _dato; }
```

No terminan en ;

Forma general de un método implementado en forma de función miembro insertada:

```
inline tipo nombre(parámetros) { código }
```

El método irá dentro de la estructura `class` y tanto los *parámetros* como el *código* son opcionales (y no figura *tipo* en los constructores y destructores, como veremos más adelante).

La palabra clave `inline` se puede omitir (la omitiremos).

Las clases en C++

Implementación de los métodos fuera de la estructura `class`

Los métodos también pueden estar implementados fuera de la estructura **`class`** (es lo más correcto):

Método **`mostrar()`**:

Operador de resolución de ámbito (::)

Nombre de la clase

Nombre del método (función miembro)

Acceso al atributo (dato privado)

```
void Clase::mostrar()
{
   cout << "El dato es: " << _dato << endl;
}
```

En el código de los métodos se puede usar todo lo que está declarado en la clase: lo público y también lo privado.
Por ejemplo, en el método **`mostrar()`** se puede usar lo siguiente: atributo **`_dato`** (privado), método accedente **`dato`** (público) y método mutador **`dato`** (público).

Los objetos en C++

Creación y uso de ejemplares

La clase ya está terminada (declarada e implementada).

Ya podemos crear ejemplares (objetos) y trabajar con ellos:

Objetos (ejemplares) de la clase

```
int main()
{
   Clase objeto1, objeto2;
   objeto1.dato(13);
   objeto2.dato(5);
   objeto1.mostrar();
   objeto2.mostrar();
   cout << objeto1.dato();
   return 0;
}
```

Pasos de mensajes

objeto • *función* (*argumentos*)

Cualquier servicio (métodos públicos)

`objeto1` y **`objeto2`** *entienden* los mismos mensajes (mismo comportamiento), pero cada uno tiene su propio **`_dato`** (valores diferentes: distinto estado)

Pasos de mensajes

Los objetos se manipulan mediante el paso de mensajes.

Cada objeto entiende tantos mensajes como servicios (métodos públicos) están definidos en su clase.

Paso de mensaje:
objeto_receptor • *función_miembro_pública* (*argumentos*)

El paso de mensaje provoca la ejecución del método con igual nombre que esté definido en la clase del objeto.

Cualquier objeto de la clase **`Clase`** entiende tres mensajes diferentes:

- ✓ **`dato`**
- ✓ **`dato`**
- ✓ **`mostrar`**

Dos métodos diferentes con el mismo nombre (son dos funciones distintas - sobrecarga).
Si se pasa un argumento entero se usa una y si no se pasa argumento se usa la otra.

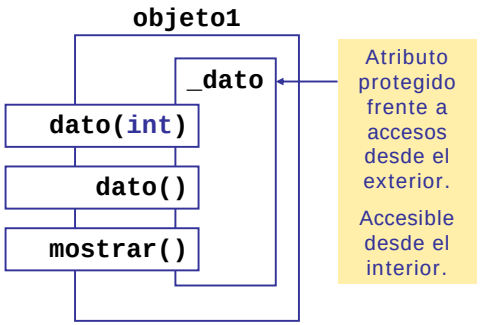
Público y privado

Los atributos deben ser privados.

Los servicios (métodos públicos), funciones miembro públicas.

A través de los objetos de una clase sólo se puede acceder (con el operador •) a lo que esté declarado como público en la clase (los servicios).

Lo privado sólo puede ser accedido directamente *desde dentro de la clase*, en el código de los métodos.



Atributo protegido frente a accesos desde el exterior.

Accesible desde el interior.

Dentro (de la clase) y fuera (de los objetos)

La clase (dentro del objeto)

Se conocen los atributos.

Se conocen todos los métodos (públicos o privados).

Implementa los métodos:

- En el método se trabaja sobre o con el objeto receptor.
- Para pasar otro mensaje al objeto receptor basta invocar el correspondiente método (sin poner ni receptor ni punto).
- El método puede utilizar también objetos locales y objetos parámetros.

Desde fuera del objeto

No se conocen sus atributos.

No se conocen sus métodos privados.

Sólo se conocen los servicios que proporciona (métodos públicos).

Se utiliza pasándole mensajes.

Paso de mensaje:

objeto.*mensaje*

Se ejecuta el código del método con igual nombre que esté definido en su clase (y con parámetros que concuerden).

Categorías de métodos

En las clases se pueden identificar distintas categorías de métodos:

- ✓ Métodos *inicializadores*: inicializan atributos.
- ✓ Métodos *accedentes*: devuelven el contenido de los atributos (cada accedente devuelve un atributo) – **`dato()`**.
- ✓ Métodos *mutadores*: establecen el contenido de los atributos (cada mutador, el contenido de un atributo) – **`dato(int)`**.
- ✓ Métodos *visualizadores*: muestran el objeto (valores de los atributos o alguna representación visual del objeto) – **`mostrar()`**.
- ✓ Métodos *computadores*: realizan cálculos y generan resultados.
- ✓ Otras categorías.

En algunas clases no tendrán sentido ciertos tipos de métodos:

- ✓ Por ejemplo, en una clase **`Pila`** no debe haber accedentes ni mutadores, ya que se trata de una *máquina de datos* que se usa sólo mediante operaciones como la inserción y la eliminación.

Una clase **`Punto`**

```
class Punto {
public:
  void inicializa() { _x = 0; _y = 0; }
  void x(double f) { _x = f; }
  void y(double f) { _y = f; }
  double x() { return _x; }
  double y() { return _y; }
  void dibujate();
private:
  double _x, _y;
};

void Punto::dibujate()
{ ... }
```

INTERFAZ

Método inicializador

Mutadores

Accedentes

Método visualizador

2 atributos

Implementación: Todo lo que no es interfaz.

Los atributos

Atributos: datos de tipos previamente definidos o ejemplares (objetos) de otras clases.

Los tipos predefinidos se pueden considerar como si fueran *clases básicas* incorporadas en el lenguaje, pero como sus variables no se utilizan exactamente de la misma forma que los objetos (pasos de mensajes), se habla a veces de *programación mixta o híbrida* (P.O.O. + Programación con tipos).

Si un atributo es un objeto de otra clase, simplemente se maneja en la implementación como tal, pasándole mensajes.

Cuando una clase A hace uso de otra clase B al utilizar en su implementación objetos de esa clase B (al pasar mensajes a atributos, parámetros o variables locales que son objetos de esa clase), se dice que la clase A es *cliente* de la clase B.

Primer tipo de relación que se establece entre las clases:

Relación de clientelismo

Una clase `Circulo`

```
class Circulo {
public:
  void inicializa()
  { _radio = 0; _centro.inicializa(); }
  void centro(Punto p) { _centro = p; }
  void radio(double f) { _radio = f; }
  Punto centro() { return _centro; }
  double radio() { return _radio; }
  void dibujate();
private:
  Punto _centro;
  double _radio;
};
void Circulo::dibujate() { ... }
```

Paso de mensaje al objeto-atributo

Objeto parámetro

Devolución de un objeto

La clase **Circulo** es *cliente* de la clase **Punto**

Atributo objeto de otra clase (**Punto**)

Convenios de denominación

Para dar nombre a los elementos de los programas se puede utilizar cualquier convenio de denominación, pero lo mejor es utilizar uno y ser consecuente. Aquí vamos a seguir las siguientes reglas:

- ✓ Los identificadores se escribirán en general en minúsculas, a excepción, en los casos que se indique, de ciertas letras.
- ✓ Las clases comenzarán por mayúscula: **`Clase`**, **`Punto`**, **`Circulo`**.
- ✓ Los atributos comenzarán por subrayado: **`_dato`**, **`_x`**.
- ✓ Los métodos comenzarán por minúscula: **`mostrar()`**, **`dato()`**.
- ✓ Lo objetos comenzarán por minúscula: **`objeto1`**, **`punto`**.
- ✓ El nombre de un método accedente será el nombre del atributo sin subrayado: **`dato()`**, **`x()`**; el método devolverá el valor del atributo.
- ✓ El nombre de un método mutador será el nombre del atributo sin subrayado: **`dato()`**, **`x()`**; el método aceptará el valor del atributo.
- ✓ Identificadores compuestos por varias palabras: la segunda y siguientes palabras llevarán su primera letra en mayúscula: **`listaObjetosGraficos`**, **`circuloColor`**.

Programas orientados a objetos

Un programa orientado a objetos está compuesto por:

- ✓ Un conjunto de clases que describen los objetos que maneja la aplicación.
- ✓ Un programa principal que pone en marcha el mecanismo de paso de mensajes.

El programa en sí se considera como un objeto aplicación, un objeto especial que crea objetos y les pasa mensajes.

```
int main()
{
  Punto p1, p2;
  // ...
  p1.y(12.3);
  p2.dibujate();
  // ...
  return 0;
}
```

El objeto aplicación crea dos objetos de la clase **Punto**: **p1** y **p2**

El objeto aplicación pasa el mensaje **y(double)** al objeto **p1**

El objeto aplicación pasa el mensaje **dibujate()** al objeto **p2**

Programas orientados a objetos

Cada paso de mensaje provoca la ejecución del correspondiente método definido en la clase del objeto receptor.

El código del método puede ser:

- ✓ Una secuencia de instrucciones que no involucre objetos (programación basada en tipos estándar)
- ✓ Una secuencia de pasos de mensajes a otros objetos (normalmente a los objetos atributos)
- ✓ Una combinación de los dos casos anteriores (programación mixta o híbrida)

Si la ejecución de un método no genera otros pasos de mensajes, el método se considera elemental.

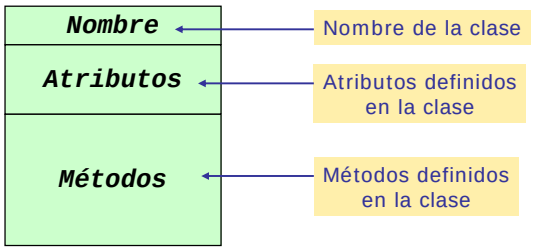
Si la ejecución de un método genera otros pasos de mensajes, se produce un encadenamiento de mensajes.

Representaciones gráficas

Para representar gráficamente los elementos de los programas orientados a objetos y sus relaciones vamos a utilizar diagramas similares a los de UML que se usan en Ingeniería del Software.

A medida que vayamos estudiando los conceptos y los mecanismos de la POO iremos viendo cómo representarlos gráficamente.

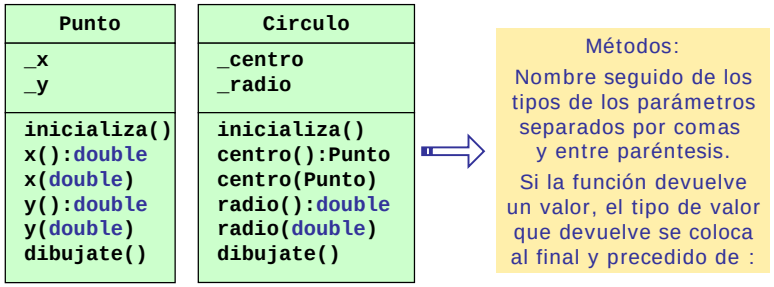
De momento, comencemos con la representación de las clases:



Representaciones gráficas

Las representaciones gráficas de las clases se pueden desarrollar con distintos niveles de detalle.

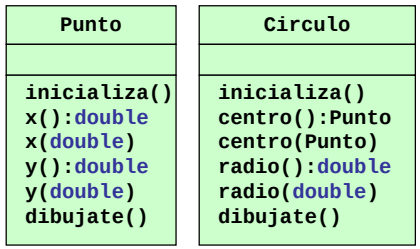
Con el mayor nivel de detalle se incluyen todas las características (atributos y métodos) definidas en la clase.



Representaciones gráficas

Si interesa se pueden omitir secciones o dejarlas en blanco.

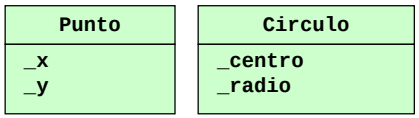
Por ejemplo, para mostrar la forma en que se usan los objetos de una clase, la parte de atributos no es relevante, por lo que la podemos dejar en blanco:



Se pueden quitar del todo secciones por el final; es decir, si la que se deja en blanco es la de métodos, al ser la última sí se puede quitar, pero como la de atributos no es la última, se dejará en blanco.

Representaciones gráficas

Si lo que se quiere es omitir la información sobre métodos, entonces se puede dejar en blanco o se puede quitar del todo:



Se pueden quitar las dos secciones de atributos y métodos, dejando simplemente el nombre de la clase; es el menor nivel de detalle:

Representaciones gráficas

Relaciones entre clases

Las relaciones entre las clases también se representan gráficamente.

La única relación entre clases que hemos visto es la de clientelismo:

Los objetos de la clase **`Circulo`** contienen objetos de la clase **`Punto`**.

La relación de clientelismo se representa conectando las clases mediante líneas con un rombo en un extremo.

El rombo se coloca en la clase que contiene objetos de la otra:

Podemos indicar el número de objetos que se contienen:

Implementación de los métodos fuera de la estructura `class`

```
class Circulo {
public:
  Punto centro();
  double radio();
  void centro(Punto);
  void radio(double);
  double circunferencia();
  double area();
  void dibujate();
private:
  Punto _centro; // atributo _centro
  double _radio; // atributo _radio
};

Punto Circulo::centro()
{ // método elemental
  return _centro;
}
// ...
```

Interfaz

Implementación

Implementación de los métodos fuera de la estructura `class`

```
double Circulo::radio()
{ // método elemental
  return _radio;
}

void Circulo::centro(Punto p)
{ // método elemental
  _centro = p;
}

void Circulo::radio(double r)
{ // método elemental
  _radio = r;
}

double Circulo::circunferencia()
{ // método elemental
  return 2 * 3.14159 * _radio;
}
```

Implementación

Implementación de los métodos fuera de la estructura `class`

```
double Circulo::area()
{ // método elemental
  return 3.14159 * _radio * _radio;
}

void Circulo::dibujate()
{
  Punto p; // objeto local al método
  int angulo, deltaX, deltaY;

  for(angulo = 0; angulo <= 360; angulo++) {
    deltaX = _radio*sin(angulo/180*3.14159);
    deltaY = _radio*cos(angulo/180*3.14159);
    p.x(_centro.x() + deltaX);
    p.y(_centro.y() + deltaY);
    p.dibujate();
  }
}
```

Implementación

Pasos de mensajes a objetos de la clase **`Punto`**

Métodos

El método **dibujate()** de la clase **Circulo**, por una parte incluye código *tradicional*, como por ejemplo:

```
for(angulo = 0; angulo <= 360; angulo++) {
  deltaX = _radio * sin(angulo / 180 * 3.14159);
  deltaY = _radio * cos(angulo / 180 * 3.14159);
```

Y por otra parte incluye pasos de mensajes:

```
  p.x(_centro.x() + deltaX);
  p.y(_centro.y() + deltaY);
  p.dibujate();
```

Se pasan los mensajes **x()**, **y()** y **dibujate()** al objeto local **p** y se pasan los mensajes **x()** y **y()** al objeto atributo **_centro**.

Así, el paso del mensaje **dibujate()** a un objeto de la clase **Circulo** provoca pasos de mensajes a objetos de la clase **Punto**.

⇒ Encadenamiento de mensajes

Encadenamiento de mensajes

Ejecución de un programa orientado a objetos

El *objeto aplicación* crea objetos y les pasa mensajes:

```
int main()
{
  Circulo c1;
  // ...
  c1.dibujate();
  // ...
```

Cada mensaje que se pasa provoca la ejecución del correspondiente método de la clase del objeto receptor.

El código del método que se ejecuta puede provocar otros pasos de mensajes (encadenamiento de mensajes).

El encadenamiento de mensajes termina al ejecutarse un método elemental (se vuelve de la función sin que haya otro paso de mensaje).

Encadenamiento de mensajes

Todos los pasos de mensajes se componen de tres elementos:

- ✓ Objeto emisor del mensaje.
- ✓ Objeto receptor del mensaje.
- ✓ Mensaje.

```
int main()
{
  Circulo c1;
  // ...
  c1.dibujate();
  // ...
```

En este caso, el objeto emisor es el *objeto aplicación*, el objeto receptor es el círculo **c1** y el mensaje es **dibujate()**.

Se ejecuta el método **dibujate()** de la clase **Circulo**, la clase del objeto receptor **c1**.

Encadenamiento de mensajes

Cuando se ejecuta el método **dibujate()** de la clase **Circulo** debido a que se ha pasado ese mensaje al objeto **c1**, el objeto emisor de los mensajes que se pasan en ese método es **c1**.

```
  p.x(_centro.x() + deltaX);
```

El objeto **c1** pasa el mensaje **x()** al objeto atributo **_centro**.

Entonces, **c1** usa el valor devuelto para generar el argumento del siguiente paso de mensaje: **c1** pasa el mensaje **x()** al objeto local **p**.

```
  p.y(_centro.y() + deltaY);
```

El objeto **c1** pasa el mensaje **y()** al objeto atributo **_centro**.

c1 pasa el mensaje **y()** al objeto local **p**, usando el valor devuelto por el mensaje anterior para generar el argumento.

```
  p.dibujate();
```

c1 pasa el mensaje **dibujate()** al objeto local **p**.

Los objetos

En los programas orientados a objetos se pueden distinguir distintos objetos, atendiendo a su existencia y su uso:

- ✓ Objetos globales: creados en el programa principal. Existen durante toda la ejecución del programa (una vez declarados).
- ✓ Objetos atributos: objetos que son atributos de otros objetos. Existen mientras existan los objetos de los que son atributos.
- ✓ Objetos parámetros: reciben los argumentos de los métodos. Existen durante la ejecución del método en cuestión.
- ✓ Objetos locales: se crean en los métodos. Existen durante la ejecución del método en cuestión (una vez creados).

Los objetos

```
class Circulo {
// ...
private:
  Punto _centro;
  double _radio;
}
void Circulo::centro(Punto p)
{
...
void Circulo::dibujate()
{
  Punto p;
...

int main()
{
  Circulo c1; Punto p1;
...
```

Objeto atributo

Objeto parámetro

Objeto local

Objetos globales

Otro ejemplo de clase: una clase **Cuenta**

Una clase para cuentas de un banco

Vamos a modelar con una clase objetos que representen cuentas de un banco. La información (atributos) de tales objetos será básica: el número de la cuenta, el saldo y el interés anual.

Como en algún momento vamos a querer mostrar información en la pantalla, lo primero es incluir la biblioteca **iostream**:

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar
```

Ahora nos centramos en la clase a desarrollar

Inicial en mayúscula

Lo primero es elegir un nombre para la clase: **Cuenta**

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar
class Cuenta {
};
```


Otro ejemplo de clase: una clase **Cuenta**

Ahora, nos centramos en los atributos, la *información* que han de contener los objetos de la clase **Cuenta**.

Establecemos un nombre y una forma (tipo/clase) para cada atributo. Colocamos la descripción de los atributos en la parte privada:

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar

class Cuenta {
private:
  long int _numero; // número de cuenta
  double _saldo;
  double _interes; // interés anual (porcentaje)
};
```

Nombres de los atributos en minúsculas y comenzando por subrayado.

A continuación nos preguntamos si resulta adecuado un método que se encargue de la inicialización de los objetos de la clase **Cuenta**.

Otro ejemplo de clase: una clase **Cuenta**

En este caso sí resultará adecuado tal método, pero concluimos que no debe servir para inicializar el atributo **_numero**, sino más bien que ese dato se proporcione al inicializar y ya no se cambie:

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar

class Cuenta {
public:
  void inicializa(long int num)
  { _numero = num; _saldo = 0; _interes = 0; }
private:
  long int _numero; // número de cuenta
  double _saldo;
  double _interes; // interés anual (porcentaje)
};
```

Nombres de los métodos en minúsculas.

Otro ejemplo de clase: una clase **Cuenta**

A continuación nos preguntamos si resulta adecuado que haya accedente y/o mutador para cada atributo. Nos parecen adecuados para **_saldo** e **_interes**, pero no para **_numero**:

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar
class Cuenta {
public:
  void inicializa(long int num)
  { _numero = num; _saldo = 0; _interes = 0; }
  double saldo() { return _saldo; }
  double interes() { return _interes; }
  void saldo(double s) { _saldo = s; }
  void interes(double i) { _interes = i; }
private:
  long int _numero; // número de cuenta
  double _saldo;
  double _interes; // interés anual (porcentaje)
};
```


Otro ejemplo de clase: una clase **Cuenta**

Luego nos preguntamos por el resto de los servicios que deben proporcionar los objetos de la clase y los incluimos como métodos:

```
#include <iostream> // entrada/salida por consola
using namespace std; // funcionalidad estándar

class Cuenta {
public:
  void inicializa(long int num)
  { _numero = num; _saldo = 0; _interes = 0; }
  double saldo() { return _saldo; }
  double interes() { return _interes; }
  void saldo(double s) { _saldo = s; }
  void interes(double i) { _interes = i; }
  void ingreso(double cantidad) { _saldo += cantidad; }
  bool reintegro(double);
  void abonoIntereses();
  void mostrar();
```

(*continúa*)

Otro ejemplo de clase: una clase **Cuenta**

```
private:
  long int _numero; // número de cuenta
  double _saldo;
  double _interes; // interés anual (porcentaje)
}; // Fin de la definición de la clase Cuenta
```

Los métodos sencillos (una instrucción de código) los hemos implementado directamente en la estructura **class**; los demás métodos los implementamos fuera:

```
bool Cuenta::reintegro(double cantidad) {
// devuelve true si hay saldo suficiente (y resta
// la cantidad); false si no hay saldo suficiente
  if(cantidad > _saldo) return false;
  _saldo -= cantidad;
  return true;
}
```

(*continúa*)

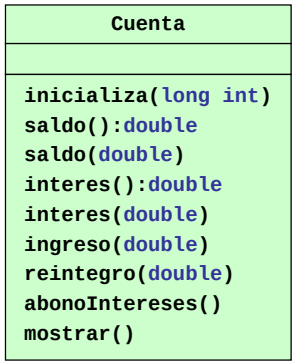
Otro ejemplo de clase: una clase **Cuenta**

```
void Cuenta::abonoIntereses() {
// abono mensual de intereses
  ingreso(_saldo * _interes / 100 / 12);
}
// el objeto se pasa un mensaje ingreso a sí mismo;
// cuando no figura delante ningún objeto receptor
// del mensaje, éste se envía a uno mismo.

void Cuenta::mostrar() {
  cout << endl;
  cout << "Número de cuenta: " << _numero << endl;
  cout << "Saldo: " << _saldo << endl;
}
```

Uso de la clase **Cuenta**

Para usar la clase basta con conocer su nombre y la forma de los métodos:



Uso de la clase **Cuenta**

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
  cc.saldo(100000);
  cc.interes(10);
  cc.mostrar();
  cc.ingreso(26000);
  cc.mostrar();
  cc.abonoIntereses();
  cc.mostrar();
  if(!cc.reintegro(10000))
    cout << "No hay saldo";
  cc.mostrar();

  return 0;
}
```

```
Número de cuenta: 24316534
Saldo: 100000

Número de cuenta: 24316534
Saldo: 126000

Número de cuenta: 24316534
Saldo: 127050

Número de cuenta: 24316534
Saldo: 117050
```

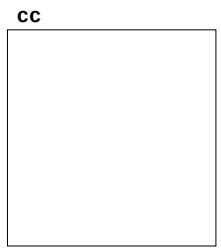
>>> Prog0101

¿Qué ocurre durante la ejecución?

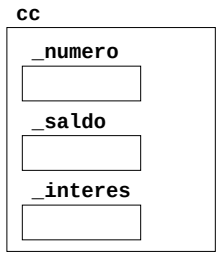
Diagramas para el análisis de la ejecución de los programas

Para el análisis de las ejecuciones de los programas utilizaremos otro tipo de diagrama para representar los objetos y ver cómo varía su estado a lo largo de la ejecución.

Cada objeto se representará como una caja identificada con el nombre del objeto en cuestión:



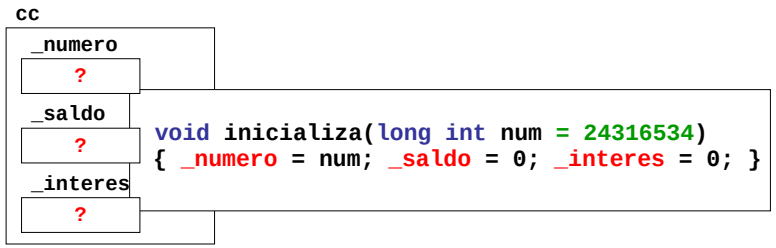
Dentro de la caja pondremos recuadros para los valores de los atributos.



¿Qué ocurre durante la ejecución?

Diagramas para el análisis de la ejecución de los programas

Cuando estudiemos la ejecución de un método colocaremos su código dentro de un recuadro que sale del objeto:



Al poner el recuadro del método de esa forma damos a entender que el método es accesible desde el exterior y puede acceder a los atributos en su implementación.

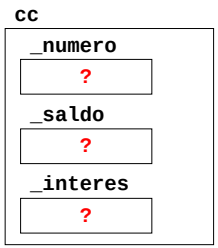
También indicaremos los valores que aceptan los parámetros.

¿Qué ocurre durante la ejecución?

Ejecución del programa paso a paso

```
int main()
{
  Cuenta cc;
```

El *objeto aplicación* crea un objeto de clase **Cuenta** y lo identifica como **cc**.

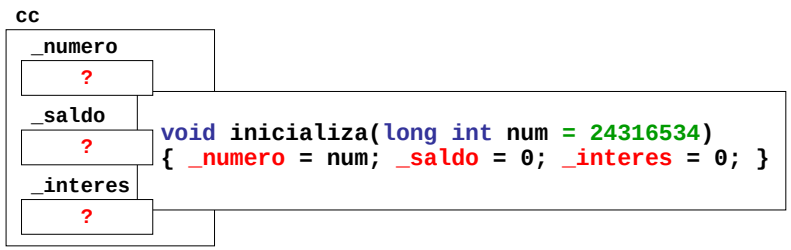


¿Qué ocurre durante la ejecución?

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
```

El *objeto aplicación* envía el mensaje **inicializa()** al objeto **cc** proporcionando el entero largo **24316534** como argumento.

Se ejecuta el método **inicializa()** de la clase **Cuenta** sobre el objeto **cc**, asignándose valores a los atributos de ese objeto **cc**.

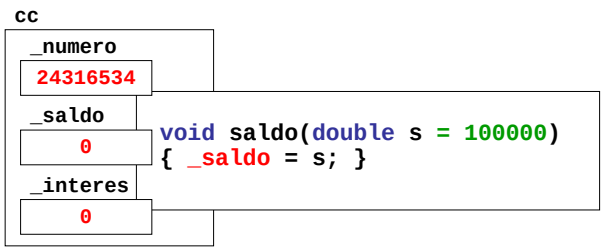


¿Qué ocurre durante la ejecución?

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
  cc.saldo(100000);
```

El *objeto aplicación* envía el mensaje **saldo()** al objeto **cc** proporcionando el valor **100000** como argumento.

Se ejecuta el método **saldo()** de la clase **Cuenta** sobre el objeto **cc**, modificándose el valor del atributo **_saldo** de ese objeto **cc**.

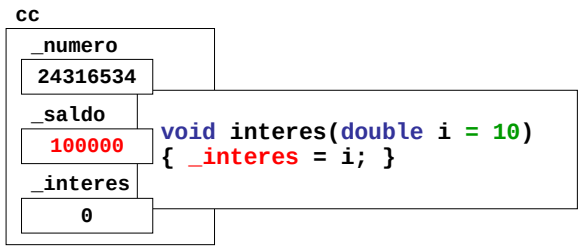


¿Qué ocurre durante la ejecución?

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
  cc.saldo(100000);
  cc.interes(10);
```

El *objeto aplicación* envía el mensaje **interes()** al objeto **cc** proporcionando el valor **10** como argumento.

Se ejecuta el método **interes()** de la clase **Cuenta** sobre el objeto **cc**, modificándose el valor del atributo **_interes** de ese objeto **cc**.

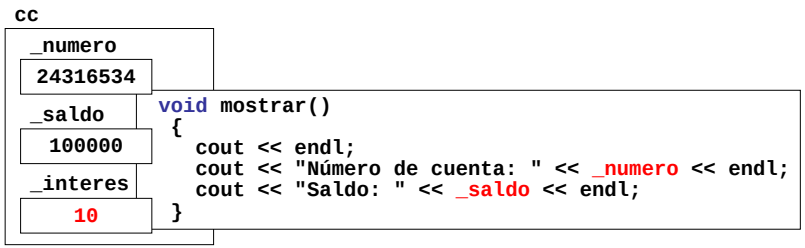


¿Qué ocurre durante la ejecución?

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
  cc.saldo(100000);
  cc.interes(10);
  cc.mostrar();
```

El *objeto aplicación* envía el mensaje **mostrar()** al objeto **cc**.

Se ejecuta el método **mostrar()** de la clase **Cuenta** sobre el objeto **cc**, mostrándose en la pantalla, entre otras cosas, los valores de los atributos **_numero** y **_saldo** de ese objeto **cc**.

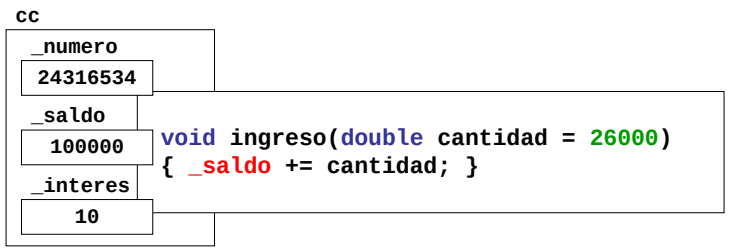


¿Qué ocurre durante la ejecución?

```
int main()
{
  Cuenta cc;
  cc.inicializa(24316534);
  cc.saldo(100000);
  cc.interes(10);
  cc.mostrar();
  cc.ingreso(26000);
```

El *objeto aplicación* envía el mensaje **ingreso()** al objeto **cc** proporcionando el valor **26000** como argumento.

Se ejecuta el método **ingreso()** de la clase **Cuenta** sobre el objeto **cc**, modificándose el valor del atributo **_saldo** de ese objeto **cc**.

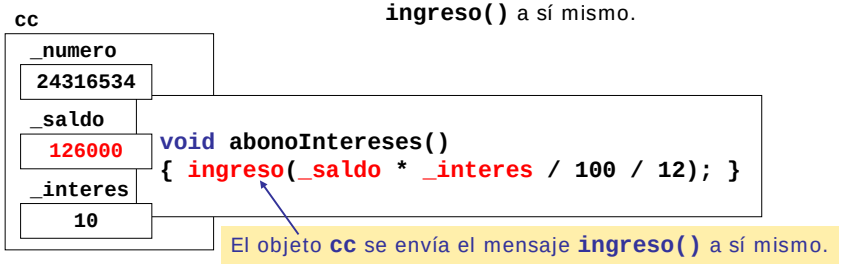


¿Qué ocurre durante la ejecución?

```
...
Cuenta cc;
cc.inicializa(24316534);
cc.saldo(100000);
cc.interes(10);
cc.mostrar();
cc.ingreso(26000);
cc.mostrar();
cc.abonoIntereses();
```

El *objeto aplicación*, después de enviar otro mensaje **mostrar()** al objeto **cc**, envía el mensaje **abonoIntereses()** al objeto **cc**.

Se ejecuta el método **abonoIntereses()** de la clase **Cuenta** sobre el objeto **cc**, lo que hace que el objeto **cc** se envíe el mensaje **ingreso()** a sí mismo.



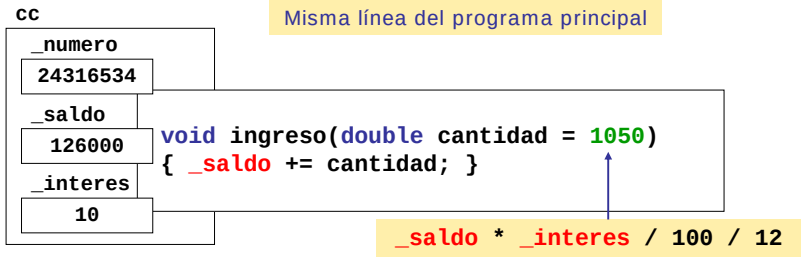
¿Qué ocurre durante la ejecución?

```
...
Cuenta cc;
cc.inicializa(24316534);
cc.saldo(100000);
cc.interes(10);
cc.mostrar();
cc.ingreso(26000);
cc.mostrar();
cc.abonoIntereses();
```

Se produce un encadenamiento de mensajes: el envío del mensaje **abonoIntereses()** al objeto **cc** provoca que el propio objeto **cc** se envíe el mensaje **ingreso()** a sí mismo.

Se ejecuta el método **ingreso()** de la clase **Cuenta** sobre el objeto **cc**, lo que modifica el valor del atributo **_saldo** del objeto **cc**.

Misma línea del programa principal

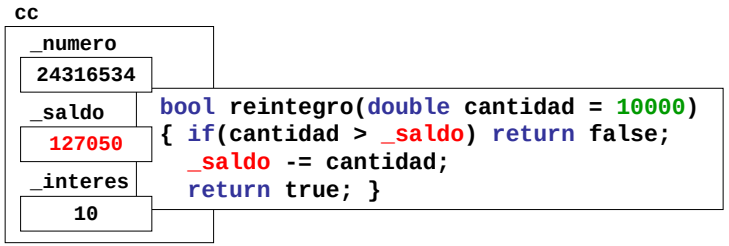


¿Qué ocurre durante la ejecución?

```
...
cc.interes(10);
cc.mostrar();
cc.ingreso(26000);
cc.mostrar();
cc.abonoIntereses();
cc.mostrar();
if(!cc.reintegro(10000))
  cout << "No hay saldo";
```

El *objeto aplicación*, después de enviar otro mensaje **mostrar()** al objeto **cc**, envía el mensaje **reintegro()** al objeto **cc**, con el valor **10000** como argumento.

Se ejecuta el método **reintegro()** de la clase **Cuenta** sobre el objeto **cc**.

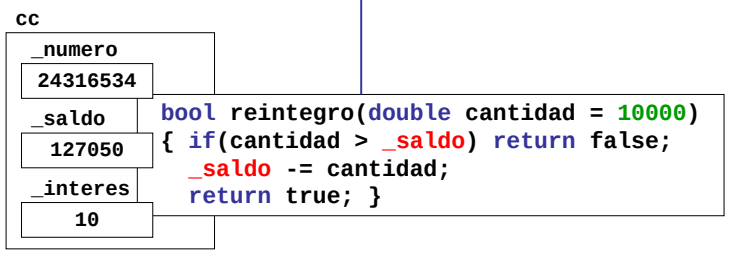


¿Qué ocurre durante la ejecución?

```
...
cc.interes(10);
cc.mostrar();
cc.ingreso(26000);
cc.mostrar();
cc.abonoIntereses();
cc.mostrar();
if(!cc.reintegro(10000))
  cout << "No hay saldo";
```

Como la **cantidad** (10000) no es mayor que el valor del atributo **_saldo**, se le resta a ese atributo la **cantidad** y se devuelve **true** como resultado de la ejecución del método.

Al volver, como la condición del **if** es falsa (**!true**), no se ejecuta su instrucción asociada.



¿Qué ocurre durante la ejecución?

```
...
cc.abonoIntereses();
cc.mostrar();
if(!cc.reintegro(10000))
  cout << "No hay saldo";
cc.mostrar();

return 0;
}
```

Tras pasar un último mensaje **mostrar()** al objeto **cc**, la función **main()** termina, devolviendo un cero al sistema operativo, indicando así que la ejecución ha transcurrido con normalidad.