

# PROGRAMACION ORIENTADA A OBJETOS



**DEPARTAMENTO de INGENIERIA ELECTRONICA y COMPUTACION**

**UNIVERSIDAD NACIONAL  
de MAR DEL PLATA**

**FACULTAD  
de INGENIERIA**



**Docentes:** Dr. Roberto M. Hidalgo (a cargo de la asignatura)  
Ing. Pablo D. Spennato (a cargo de la práctica)  
Ing. Marco L. Viola

Página de la cátedra: <https://sites.google.com/site/poofimdp>

## Reglamento de la Cátedra

- Se tomarán 2 (dos) exámenes Parciales de contenido teórico-práctico (**P1** y **P2**) y un único Parcial Flotante (se podrá recuperar sólo uno de los Parciales).
- **Aprobación**: la nota del Primer Parcial no debe ser inferior a 6 (seis) puntos y la nota del Segundo no debe ser inferior a 7 (siete) puntos: ( **$P1 \geq 6$** ) && ( **$P2 \geq 7$** ).
- **Habilitación**: se deben cumplir las siguientes condiciones a) La nota del Segundo Parcial es menor que 7 (siete) pero no menor que 5 (cinco); b) la nota del Primer Parcial es de por lo menos 5 (cinco) y c) la suma de ambas notas debe ser al menos 11 (once) puntos: ( **$P1 \geq 5$** ) && ( **$7 > P2 \geq 5$** ) && ( **$(P1 + P2) \geq 11$** ).
- El examen Totalizador (final) consistirá en la realización de un programa que cumpla con determinadas características que serán fijadas por la cátedra (ver la sección correspondiente al Reglamento en la página alojada en el Campus de la Facultad).
- Los alumnos que tengan una **nota inferior a 5** (cinco) en el **Segundo Parcial desaprueban** la materia independientemente de la nota que hayan obtenido en el Primer Parcial, debiendo recursar la misma al próximo año.
- **No existe la posibilidad de brindar una Recursada en el primer cuatrimestre.**



# Contenidos

**Programación Orientada a Objetos:** Conceptos básicos – Programación Procedural vs. POO – Relaciones y diferencias entre los lenguajes C y C++ – Variable por referencia – Nociones de Encapsulamiento, Herencia y Polimorfismo – Clases y Objetos – Mensajes entre objetos – Tipos de relaciones: asociación, agregación y composición – Diagrama de clases en UML.

**Lenguaje de Programación C++:** Espacios de nombres (*namespaces*) – Entrada y salida de datos (*streams*) – Manejo de archivos – Gestión dinámica de memoria (operadores *new*, *delete*) – Constructores y destructores – Modificadores de almacenamiento – Clase *string*.

**Encapsulamiento y Polimorfismo:** Encapsulamiento, funciones *getters* y *setters* – Sobrecarga de funciones (funciones *friend*) – Sobrecarga de operadores – Puntero *this*.

**Herencia:** Concepto de herencia – Herencia simple y múltiple – Interfaces.

**Polimorfismo:** Concepto de polimorfismo – Polimorfismo y funciones virtuales – Clases abstractas – Binding dinámico.

**Plantillas y excepciones:** Plantillas (*templates*) – Manejo de Excepciones – C++ STL (Standard Template Library) – Colecciones.

**Lenguaje Java:** Similitudes y diferencias respecto a C++ – Tipos de datos – Acceso a funciones de librería (*import*) – Entrada y salida de datos – Excepciones.

**Interfaz gráfica:** GUI (*Graphics User Interface*) – Controles – Diálogos modales y no modales – Menús – Barras de herramientas – Librerías gráficas – Eventos – Manejadores de eventos.

## Cronograma

Nº	Fecha	Tema	Nº	Fecha	Tema
1	17/08	Presentación. C vs. C++. Introducción a la POO.	17	19/10	Manejo de excepciones.
2	20/08	Clases y objetos. Diagrama de clases en UML.	18	22/10	Práctica N° 6.
3	24/08	Práctica N° 1.	19	26/10	Plantillas. C++ Standard Template Library.
4	27/08	Namespaces. Streams. new/delete. Const/Dest.	20	29/10	Práctica N° 6 (continuación).
5	31/08	Práctica N° 2.	21	02/11	Colecciones.
6	03/09	Encapsulamiento. Sobrecarga de func. y operad.	22	05/11	Práctica N° 6 (finalización).
7	07/09	Práctica N° 3.	23	09/11	Lenguaje Java. Comparaciones con C++.
8	10/09	Práctica N° 3 (continuación).	24	12/11	Práctica N° 7.
9	14/09	Herencia simple y múltiple. Interfaces.	25	16/11	Práctica N° 7 (continuación).
10	17/09	Práctica N° 4.	26	19/11	Práctica N° 7 (finalización).
11	24/09	Práctica N° 4 (continuación).	27	23/11	GUI- Librerías. Controles. Contenedores. Eventos.
12	28/09	Clase de reserva.	28	26/11	Práctica N° 8.
13	01/10	<b>PRIMER PARCIAL.</b>	29	30/11	<b>SEGUNDO PARCIAL</b>
14	05/10	Polimorfismo (func. virtuales). Cs. abstractas.	30	03/12	Clase de consulta.
15	12/10	Práctica N° 5.	31	07/12	<b>RECUPERATORIO FLOTANTE</b>
16	15/10	Práctica N° 5 (continuación).			



# Bibliografía

## ➤ Básica

**“The C++ programming language (Third Edition)”**

Stroustrup Bjarne. Addison Wesley.

**“Programming principles and practice using C++”**

Stroustrup Bjarne. Addison Wesley.

**“Thinking In C++ Vol. 1 / Vol. 2 (Second Edition)”**

Bruce Eckel. Prentice Hall.

**“C++: The complete reference (Third Edition)”**

Herbert Schildt . McGraw-Hill.

**“C++ Primer (Fourth Edition)”**

Stanley B. Lippman. Addison Wesley.

**“Guía de iniciación al Lenguaje JAVA”**

<http://pisuerga.inf.ubu.es/lsi/Invest/Java/Tuto/Index.htm>.

# Bibliografía

## ➤ Complementaria

### **“C++ estándar”**

British Standards Institute. Anaya Multimedia.

### **“Aprenda C++ como si estuviera en Primero”**

J. García de Jalón, J. I. Rodríguez , J. M. Sarriegui , A. Brazález.

Universidad de Navarra. <http://www1.ceit.es/asignaturas/informat1/ayudainf/index.htm>

### **“Programación orientada a objetos”**

Luis Joyanes Aguilar. McGraw Hill.

### **“Introducción a la OOP”**

F. Moreno. Grupo EIDOS.

<http://www.scribd.com/doc/18773409/Intoduccion-a-la-OOP-Grupo-EIDOS>

### **“Object-Oriented Programming in C++ (Fourth Edition)”**

R. Lafore. Sams Publishing.

### **“C++ Primer Plus (Fourth Edition)”**

S. Prata. Sams Publishing. 2001.

### **“Piensa en Java (Third Edition)”**

B. Eckel. Pearson Educación S.A. 2002.



# Paradigmas de Programación

Según los conceptos en que se basa un lenguaje de programación se tienen distintas maneras y estilos para plantear la solución.

- Procedurales / Orientados a Objetos: se basan en la **abstracción** de los tipos de datos. Se trata de representar las características variables de los objetos mediante tipos que el compilador pueda tratar (números enteros o caracteres alfanuméricos). El programa será una colección de algoritmos que opere sobre los datos modelados.

## Programación Procedural

- Código y datos separados, sin ninguna conexión formal
  - ✓ código → funciones
  - ✓ datos → estructuras
- Programa: es un conjunto de llamadas a funciones (secuencia de instrucciones que describen la solución)

## Programación Orientada a Objetos

- Todos **objetos** descritos por características y acciones.
  - ✓ características → datos (**atributos**)
  - ✓ acciones → funciones que operan sobre los datos → (**métodos**)
- Programa: conjunto de objetos que se relacionan entre sí “comunicándose” a través de mensajes

# Paradigmas de Programación

- Funcionales: eliminan la idea de tipo de datos, los tratan como símbolos y hacen hincapié en las operaciones que pueden aplicarse sobre estos símbolos, agrupados en listas o árboles. Se emplea únicamente el concepto de función aplicado a símbolos, siendo una de sus características principales el empleo de las funciones recursivas (LISP. 2).
- Lógicos: trabajan directamente con la lógica formal, se trata de representar relaciones entre conjuntos, para luego poder determinar si se verifican determinados predicados. El lenguaje lógico más extendido es el Prolog.



# Programación Procedural

También llamada programación **imperativa** (C). La idea básica es definir los algoritmos o procedimientos más eficaces para tratar los datos del problema. Involucra los conceptos de:

- Tipos de datos: pueden ser elementales o compuestos (arreglos y estructuras).
- Operadores y expresiones: incluyendo los operadores dot (.) y arrow (→).
- Algoritmo: conjunto de operaciones (reglas) bien definidas para resolver un problema en un número finito de pasos. Diseño top-down para resolver problemas complejos.
- Estructuras de control: secuenciales, condicionales o de selección y de repetición (bucles).
- Funciones y procedimientos: se deben conocer sus parámetros y valor de retorno, no importa la manera en la que se programaron (**abstracción**).
- Constantes y variables: globales y locales. Las variables son creadas al entrar en su ámbito y eliminadas al salir de él.
- Programación modular: módulo = conjunto de procedimientos y datos interrelacionados. Se recurre al principio de ocultación de información (los datos contenidos en un módulo no son accesibles desde el exterior). La comunicación entre módulos se realiza a través de procedimientos públicos definidos por el programador. Un módulo bien definido podrá ser reutilizado y su depuración será más sencilla al tratarlo de manera independiente.

## Tipo Abstracto de Datos (TAD)

- Un TAD es un **encapsulamiento** que contiene la definición de un nuevo tipo de datos y todas las operaciones que se pueden realizar con él (*matriz, string, complejo*).
- El programa trata al TAD como un tipo elemental, accediendo a él sólo a través de los operadores que se hayan definido.
- Se facilitan las modificaciones e incorporaciones de nuevas características o funciones.
- C no está preparado para trabajar con TAD. Las variables de un nuevo tipo (estructuras), deben ser pasadas como parámetros a los procedimientos específicos.
- C++ incorpora mecanismos de definición de tipos por parte del usuario que se comportan casi igual que los tipos propios del lenguaje (se pueden aplicar los mismos operandos).



# POO – Introducción

- La solución se basa en el empleo de **objetos** (**módulos**, **componentes**) como elementos fundamentales. Un objeto es una abstracción de algún hecho o ente del mundo real.
- **Objeto** = **atributos** + **métodos**.
  - **atributos** (datos) = representan las características o propiedades del elemento.
  - **métodos** (funciones) = fijan su comportamiento o actividad.
- Los objetos pueden ser diseñados y comprobados de manera independiente del programa que va a usarlos y por lo tanto podrán ser reutilizados en otros programas.
- En lugar de un conjunto de rutinas que operan sobre datos, se tiene objetos que interactúan pasándose **mensajes** y **respuestas**.
  - **Evento** = suceso en el sistema (generado por el usuario o por un objeto). El sistema maneja el evento enviando un mensaje específico al objeto relacionado.
  - **Mensaje** = comunicación dirigida a un objeto para que se ejecute uno de sus métodos, con parámetros determinados por el evento que los generó.

# POO – Ventajas

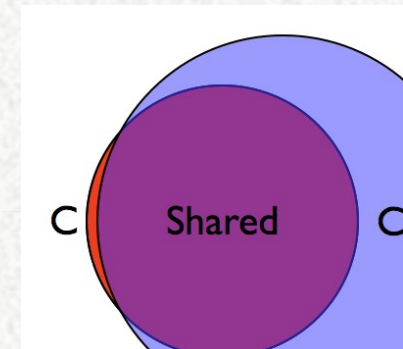
- Reutilización de código:
  - ✓ Ahorra tiempo en el desarrollo de programas.
  - ✓ Empleo de software que ya ha sido probado.
- Fácil mantenimiento y depuración de los programas.
- Extensibilidad: posibilidad de ampliar la funcionalidad de la aplicación de manera sencilla.
- Modularidad y encapsulamiento: el sistema se descompone en objetos con responsabilidades claramente especificadas.

Observación de ingeniería de software: es importante escribir programas que sean **comprensibles** y **fáciles de mantener**. Las modificaciones son la regla más que la excepción. Las **clases** facilitan la capacidad de modificación de los programas.



# LENGUAJE DE PROGRAMACION C++

- **C++** = **Cpp** (C plus plus) = incremento de C = **C with classes**. Desarrollado en 1980 por Bjarne Stroustrup en ATT (American Telephone and Telegraph), la misma compañía en la que en 1972 Brian Kernigham y Dennis Richie desarrollaron C.
- C++ mantiene las ventajas de C (riqueza de operadores y expresiones, flexibilidad, concisión y eficiencia) eliminando algunas dificultades y limitaciones (manejo de excepciones, sobrecarga de operadores y uso de templates o plantillas). Algunas modificaciones introducidas en C++ fueron adoptadas por el ANSI C99 para compiladores C.
- La evolución de C++ continuó con la aparición de **Java** (aplicaciones en Internet).



Reglas para la sintaxis de cada sentencia (uso general en todas las publicaciones y ayudas): a) los valores entre corchetes "[ ]" son **opcionales** (salvo vectores), b) el separador "|" delimita las **opciones** que pueden elegirse. Los valores entre "<>" se refieren a **nombres**. Los textos **sin delimitadores** son de aparición **obligatoria**.

# Modificaciones menores de C++ respecto a C

- **Nombre de los archivos fuente:** nombre.c → nombre.cpp (extensión → compilador).
- **Comentarios:** /\* puede abarcar varias líneas \*/  
// una sola línea (adoptado también por el ANSI C)
- **Declaración de variables:** en C debe efectuarse al inicio de la función, antes de la primer sentencia. En C++ pueden declararse dentro del bloque en que se utilizan, antes de asignarles un valor (adoptado por el ANSI C99).
- **Ámbito de las variables:** temporal (intervalo de tiempo en el que un objeto existe) y de acceso (alcance o visibilidad). Las funciones (y otros objetos), también tienen distintos ámbitos. C++ dispone de especificadores de tipo de almacenamiento (auto, extern, static, register) y de modificadores (const, volatile y mutable).
  - **auto:** almacenamiento automático. Es el modificador por defecto cuando se declaran variables u objetos locales. Su valor se pierde al salir de su ámbito (bucle o función).



# Modificaciones menores de C++ respecto a C

## ➤ **Ámbito de las variables:**

- **static**: se asigna una dirección de memoria fija para el **objeto** (ámbito temporal total). El ámbito de acceso es local. El valor inicial es nulo por defecto y mantiene el último valor asignado. Las **funciones estáticas** sólo son accesibles desde el fichero que las declara.
- **const**: indica que el valor del **objeto** no puede ser modificado. Debe ser inicializado en la declaración (C++ no permite dejar una constante indefinida). La ventaja respecto de una macro, es que se pueden validar las operaciones por tipo de variable.
  - **const tipo** variable = inicialización;      // {val1, .... valn} para vectores/estructuras
  - **tipo** func(**const tipo** \*vble);    // **no se puede** modificar vble (aunque tiene su dirección)
- **volatile**: el objeto puede ser modificado mediante procesos externos (programas multihilo interrupciones o registros con periféricos). Se asegura que el sistema operativo consulte su valor actual antes de usarlo (el valor puede cambiar sin que el programa se entere). Cuando se utiliza un puntero y se le quiere asignar la dirección de una variable volatile en su declaración se debe incluir expresamente el modificador (**volatile int** \*yptr = &y).

# Modificaciones menores de C++ respecto a C

- **Enumeraciones (*enum*)**: es preferible su uso respecto a *define*; se mejora la claridad y facilita de comprensión de los programas, los valores se mantienen agrupados y por defecto adoptan valores consecutivos. Se **crea un nuevo tipo de variable** (en C sólo *int*), cuyo **nombre pasa a ser palabra reservada**. Existe una comprobación de tipos más estricta.

```
enum color {ROJO, VERDE, AZUL, AMARILLO};    // ROJO = 0, VERDE = 1, AZUL = 2, AMARILLO = 3
```

```
enum color {ROJO = 3, VERDE = 5, AZUL = 7, AMARILLO};    // AMARILLO = 8
```

- ✓ declaración de las variables:

```
enum color pintura, fondo;    /* esto es C */  
color pintura, fondo;    // es C++ (se omite la palabra clave enum)
```

- ✓ asignación de valores:

```
fondo = VERDE;    // en C es un int, en C++ es un color  
fondo++;    // en C fondo adopta el valor 6  
            // en C++ no se puede (el ++ se aplica sólo en un tipo int)  
fondo = (color)25;    // debe realizarse el cast
```



# Modificaciones menores de C++ respecto a C

- **Estructuras**: no es necesario utilizar la palabra *struct*. Se crea un nuevo tipo de datos.

```
struct datos
{
    long doc;
    char nombre[30];
};
```

```
struct datos alumno;          /* en C, para omitir struct: typedef struct {...} datos; */
datos alumno;                  // en C++ se creó el nuevo tipo datos
```

- **Conversión explícita de tipo**: además del **cast** utilizado en C (`nuevo_tipo`)`otro_tipo`, se puede utilizar la forma alternativa *nuevo\_tipo(valor)* (como si fuera una función).

```
double z;
int x = 5, y = 2, j;
```

```
z = (double)25;                /* forma compatible en C y C++ */
z = double(25);                // forma sólo admitida en C++
j = int(x/y);                  // forma en C++ (como si se invocara una función)
```

# Palabras reservadas

- C++ introduce 15 nuevas palabras reservadas:

<b>catch</b>	<b>class</b>	<b>delete</b>	<b>friend</b>	<b>inline</b>
<b>new</b>	<b>operator</b>	<b>private</b>	<b>protectec</b>	<b>public</b>
<b>template</b>	<b>this</b>	<b>trow</b>	<b>try</b>	<b>virtual</b>

- El ANSI añadió al estándar de C++ nuevas palabras para soportar tipos adicionales (bool y wchar\_t), nuevas características (namespace) e identificación de tipos en tiempo de ejecución:

<b>bool</b>	<b>const_cast</b>	<b>dynamic_cast</b>	<b>false</b>
<b>mutable</b>	<b>namespace</b>	<b>reinterpret_cast</b>	<b>static_cast</b>
<b>typeid</b>	<b>using</b>	<b>true</b>	<b>wchar_t</b>



## POO – Características

- **Modularidad:** permite subdividir una aplicación en partes más pequeñas (módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos.
- **Abstracción:** permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades.
- **Encapsulamiento:** cada objeto está aislado del exterior. Cada tipo de objeto expone una **interfaz** a otros objetos donde se presenta el prototipo de las funciones que puede realizar, el usuario no necesita conocer la implementación. Facilita el mantenimiento y depuración de los programas. Ante una variación en el código de la clase los usuarios no necesitarán cambiar sus líneas de código si no cambia su interfaz.
- **Polimorfismo:** permite utilizar el mismo nombre para funciones de comportamientos diferentes (asociadas a objetos distintos), por ejemplo una función *suma()* que calcule la suma de matrices o de números complejos. Cuál de los métodos será invocado se resuelve en *tiempo de ejecución* cuando un objeto particular lo necesite. Esta característica se denomina **asignación tardía** o **dinámica**. Algunos lenguajes proporcionan también medios más **estáticos** (*tiempo de compilación*): plantillas (**templates**) y **sobrecarga de operadores**.

## POO – Características

- **Herencia:** permite definir nuevas clases (subclase o derivada) en base a otras ya existentes (base o superclase). Una subclase **hereda** todos los métodos y atributos de su superclase, además de poder definir miembros adicionales (ya sean datos o funciones). Las subclases también pueden redefinir (**override**) métodos definidos por su superclase: responden al **mismo mensaje** que su superclase con su propio método (**polimorfismo**).
- **Cohesión y acoplamiento:** el principal objetivo al plantear el diseño de una aplicación es minimizar el costo de mantenimiento, por los cambios que surgen en el sistema. Un diseño efectivo minimiza la probabilidad de que se propaguen los cambios. Se debe **reducir** el **acoplamiento** entre componentes y **aumentar** la **cohesión** interna de cada uno.
  - Acoplamiento: grado de interdependencia que hay entre los distintos módulos de un programa. Dos elementos están acoplados (no deseable) en la medida en el que los cambios en uno tienden a necesitar cambios en el otro. Un acoplamiento simple se presenta cuando un componente accede directamente a un dato que pertenece a otro.
  - Cohesión: medida de fuerza o relación funcional existente entre las sentencias o grupos de sentencias de un mismo módulo. Un módulo cohesionado (deseable) ejecutará una única tarea sencilla interactuando muy poco o nada con el resto de módulos del programa. Un elemento puede tener poca cohesión tanto por ser muy grande o muy pequeño.



# POO – Clases y Objetos

- **Clase (class)**: es un **tipo de dato definido por el usuario (TAD)**. Se puede considerar como una generalización de las *estructuras* de C.
- **Objetos, instances** (instancias) o **data objects**: son las variables concretas que se crean de una determinada **clase**, la cual describe su estructura (información y comportamiento), mientras que el estado de la instancia se define por las operaciones ejecutadas.
- Un objeto tiene su propio conjunto de **datos** o **variables (atributos)** y **funciones (métodos)** **miembro**. Para proteger a las variables de modificaciones no deseadas se recurre al concepto de *encapsulamiento, ocultamiento o abstracción de datos*.
- Especificadores de acceso a los miembros de la clase:
  - **public** = se pueden acceder libremente desde fuera de la clase. Los métodos y atributos públicos forman la **interfaz** del objeto, a través de la cuál se comunica con otros objetos.
  - **private** = sólo pueden ser accedidos por las funciones y operadores de la misma clase.
  - **protected** = son privados para las funciones externas pero públicos para las clases derivadas.

# POO – Clases y Objetos

## Definición de métodos

Los **métodos** de una clase se definen anteponiendo a su nombre el nombre de la clase y el *operador de resolución de visibilidad* (**:: = scope resolution operator**), de la forma:

*tipo\_dato\_retorno* <nombre\_clase> :: <nombre\_metodo>(parametros) { .....; }

Mediante esta notación se puede distinguir entre funciones que tengan el mismo nombre pero distintas visibilidades y permite también acceder a funciones desde puntos del programa en que éstas no son visibles.

En general se crea un archivo *clase.h* que contiene la definición de la clase y un archivo *clase.cpp* con las definiciones de las funciones y operadores (implementación de la clase), al cual el usuario no necesita acceder. Finalmente el archivo *main.cpp* contiene un programa principal que utiliza algunas de las posibilidades de la nueva *clase*.

## Acceso a los miembros públicos

Se acceden mediante los operadores punto (.) ó flecha (→) luego del nombre del objeto.



## POO – Clases y Objetos (ejemplo)

```
class <datos>                // declaración de una clase con 2 enteros y una función públicos
{
    public:
        int a, b;
        double func(int);
        void setFpriv(float);
        float getFpriv();
    private:
        float fpriv;
};
```

```
datos objeto1,                // declaración de un objeto de la clase datos
    *objeto2;                  // declaración de un puntero a un objeto de tipo datos
```

```
int c = objeto1.a;             // accede al miembro público a de objeto1
int d = objeto2→b;             // accede al miembro público b de objeto2
double valor1 = objeto1.func(d); // accede a la función pública de objeto1
double valor2 = objeto2→func(c); // accede a la función pública de objeto2
float valor3 = objeto1.getFpriv(); // accede al valor privado de objeto1
objeto2 → setFpriv(valor3);     // fija el valor privado de objeto2
```



# DIAGRAMAS EN UML

**UML** son las siglas de **Unified Modeling Language** (Lenguaje Unificado de Modelado). No es un lenguaje de programación propiamente dicho, sino una serie de **normas** y **estándares gráficos** respecto a cómo se deben representar los esquemas y diagramas relativos al software. Ha sido adoptado a nivel internacional por numerosos organismos, empresas (Texas Instruments, Oracle, Microsoft) y equipos de desarrollo de software para planificar y documentar cómo se construyen los programas informáticos complejos. En general los usuarios individuales y los equipos de 2 ó 3 personas no usan estas herramientas, aunque su empleo facilita y mejora la documentación.

Define normas para construir **muchos** tipos de esquemas, por lo que es usado por analistas funcionales (definen qué debe hacer un programa sin escribir el código) y analistas programadores (lo estudian el problema y escriben el código para resolverlo). UML no se encuentra asociado a un lenguaje de programación, los diagramas pueden ser implementados en C++, Java, C#, Python, etc.



# UML – Tipos de diagramas

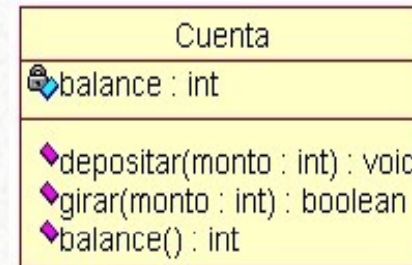
- de **clases**: para UML una clase es una entidad. Puede ser un diagrama o representación de conceptos que intervienen en un problema, o un diagrama de clases software (class). El sentido de se lo da la persona que lo construye.
- de secuencia: se usan para representar objetos de software y el intercambio de mensajes entre ellos, representando la aparición de nuevos objetos de izquierda a derecha. Se puede utilizar también para indicar cómo evolucionan los métodos en el tiempo.
- de estados: indican cómo evoluciona un sistema (cómo va cambiando de estado) a medida que se producen determinados eventos.
- de colaboración: representan objetos o clases y la forma en que se transmiten mensajes y colaboran entre ellos para cumplir un objetivo.
- de casos de uso: representan a los actores y casos de uso (procesos principales) que intervienen en un desarrollo de software.
- Otros diagramas: de actividad, de paquetes, de arquitectura de software, etc.

# UML – Relaciones entre Clases

- **Clases**: atributos, métodos y visibilidad.
- **Relaciones**: herencia, asociación, agregación, composición y uso.

Clase: representada por un rectángulo que posee tres divisiones reservadas para:

- ✓ Superior: el nombre de la clase.
- ✓ Intermedio: las propiedades o atributos.
- ✓ Inferior: los métodos u operaciones.



Atributos y métodos: de acuerdo al grado de visibilidad con el entorno pueden ser:

- ✓ **public** (+,

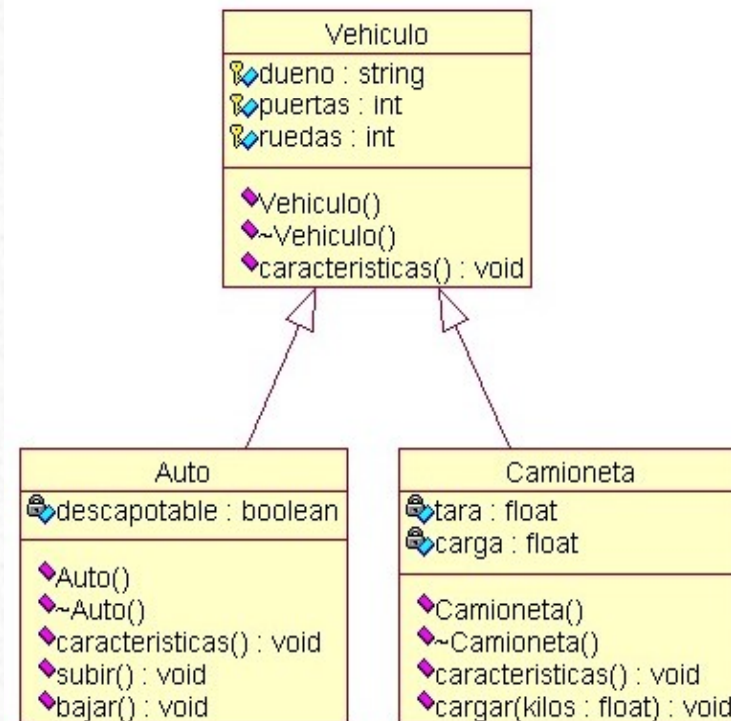


# UML – Relaciones entre Clases


- **Herencia (Especialización/Generalización)**  $\longrightarrow \triangleright$  : indica que una **subclase** hereda los métodos y atributos de una **superclase**. Por lo tanto, además de poseer sus propios métodos y atributos, también poseerá los que sean visibles en la superclase (**public** y **protected**).

## Ejemplo:

Las clases *Auto* y *Camioneta* heredan de la clase *Vehículo*; por lo tanto un objeto *Auto* posee los atributos protegidos de *Vehículo* (*dueno*, *puertas*, *ruedas*) y además una característica particular que es *descapotable*. *Camioneta* también hereda las características de *Vehículo* (*dueno*, *puertas*, *ruedas*) y posee como particularidad propia *tara* y *carga*.



# UML – Relaciones entre Clases

➤ **Asociación**  : permite indicar que dos clases están asociadas de algún modo; que puede ser:

- Agregación
- Composición
- Dependencia.


En UML, se puede indicar la **cardinalidad** de las relaciones; que indica el grado y nivel de dependencia. Se anotan en cada extremo de la relación y éstas pueden ser:

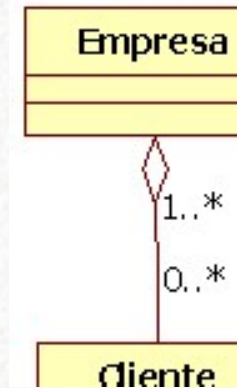
- ✓ **uno o muchos**: **1..\*** (1..n) Por ejemplo, un cliente puede ter asociadas muchas Ordenes de compra, en cambio una Orden de Compra sólo puede tener asociado un Cliente.
- ✓ **0 o muchos**: **0..\*** (0..n).
- ✓ **número fijo**: **m** (m denota el número).




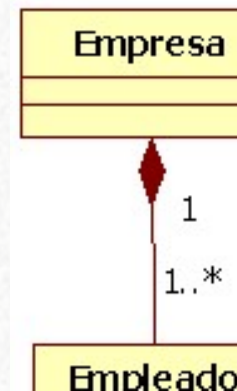


# UML – Relaciones entre Clases

- **Agregación**  : es un tipo de asociación que indica que una clase es parte de otra clase (composición débil). El objeto base utiliza al incluido para su funcionamiento. La relación es dinámica, el tiempo de vida del objeto incluido es independiente del que lo incluye. Se representa mediante un rombo de color blanco colocado en el extremo en el que está la clase que representa el “todo”. Por ejemplo, una Empresa agrupa a varios Clientes.



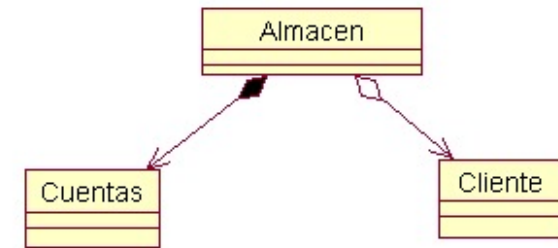
- **Composición**  : es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora. La relación es estática, la supresión del objeto compuesto conlleva la supresión de los componentes. Se representa mediante un rombo de color negro colocado en el extremo en el que está la clase que representa el “todo”. Por ejemplo, un objeto Empresa está compuesto por uno o varios objetos del tipo Empleado. El tiempo de vida de los objetos Empleados depende del tiempo de vida de Empresa.



# UML – Relaciones entre Clases

## Ejemplo combinado agregación y composición:

Un almacén posee Clientes y Cuentas. Cuando se destruye el objeto Almacén también son destruidos los objetos Cuentas asociados, mientras que los objetos Cliente no son afectados. Para Cuentas se realiza una Composición y con Cliente se establece una Agregación.



- **Dependencia** o **Instanciación (uso)** -.-.-.-> : es una relación de uso entre dos clases (una usa la otra). Es la relación más básica entre clases (más débil). Se representa por una flecha discontinua cuyo sentido indica quien usa (ClaseA) a quien (ClaseB), ya sea instanciándola o recibéndola como parámetro de entrada en uno de sus métodos. Todo cambio en la ClaseB podrá afectar a la ClaseA. Además, la ClaseA conoce la existencia de la ClaseB, pero ésta desconoce a la ClaseA. Por ejemplo, una Aplicación gráfica que instancia una Ventana (la creación del objeto Ventana está condicionada a la instanciación proveniente desde el objeto Aplicación).

