Unidad Didáctica 5:

Introducción a la Programación Orientada a Objetos

- **Clases**
- **Estructuras con funciones miembros.**
- **Definición de una clase.**
- **Constructores.**
- **Tipos de constructores.**
- **Sobrecarga de funciones, (polimorfismo).**
- **Definición de una función miembro fuera de la clase.**
- **Especificadores de acceso, (private, public, protected)**

En esta sección abordaremos los conceptos básicos - iniciales, de la Programación Orientada a Objetos, (POO).
Se verán algunos conceptos como clases, polimorfismo, herencia, etc., directamente con ejemplos, sin utilizar muchas definiciones "académicas", tan sólo ejemplos.

Clases:

En lenguaje C tradicional existen las estructuras de datos, las cuales se definen con la palabra clave **struct**, ejemplo:

```
struct Coordenadas
{
   int x;
   int y;
   int z;
}
```

Con una estructura uno crea un tipo de dato nuevo, en este caso, se puede declarar una variable de tipo Coordenadas, la cual puede almacenar 3 valores enteros:

```
struct Coordenadas coo;    //Declaración de la variable coo de tipo Coordenadas
coo.x=7;                   //Valores iniciales para los datos miembros.
coo.y=15;
coo.z=55;
```

x, y, z son los *"datos miembros"* de la estructura. Para manipular estos datos, (asignarles un valor inicial, cargarlos, mostrarlos, etc.), uno puede escribir funciones globales en su programa. Ejemplo:

void Carga(void)
void Muestra(void)

Bueno, se podría decir que una estructura es el "antepasado" más directo de una clase. ¿Por qué?.
Que tal si las funciones con las cuales uno manipula los datos de la estructura formaran parte de ella, o sea, una estructura tal que además de definir sus datos miembros también definiera las funciones para manipularlos.

Veamos lo anterior en pseudocódigo:

```
Clase Coordenadas
{
        publicas int x,y,z;

        coordenadas:Cargar()
        {
                x=8;
                y=6;
                z=3;
        }

        Coordenadas:Mostrar()
        {
                Escribir ("X=",x)
                Escribir ("Y=",y)
                Escribir ("Z=",z)
        }
}

Programa Principal
 {
        Coordenadas punto=Nueva Coordenadas()
        punto.Cargar();
        punto.Mostrar();
}
```

¿Encontró las diferencias?.
La verdad, no son muchas. En lugar de struct se pone clase, luego se agrega la etiqueta publicas, antes de definir las funciones miembros, ya que para **una estructura** los datos miembros y funciones miembros son **por defecto públicos**, pero **en una clase** por defecto los datos miembros son **privados**, (esto forma parte, entre otras cosas, de lo que se llama "encapsular"), y sólo las funciones públicas pueden tener acceso a los datos privados.

En la POO, utilizando clases, ya no se habla de "definir" una variable de una clase en particular, sino que se crea una "instancia" o un objeto de dicha clase.

¿Por qué usar clases y no estructuras?.

A veces la diferencia, aparte de la sintaxis, no es del todo "pesada" como para justificar una clase. En este ejemplo no hacía falta definir una clase, la versión de la estructura es más que suficiente.
Pero cuando el concepto del objeto a crear es un tanto más complejo, y preocupa, por ejemplo, la protección de los contenidos de los datos miembros, o se tiene una gran cantidad de funciones miembros, o simplemente se pretende en serio programar según POO, es cuando una clase se hace presente.
Pues como supongo astutamente dedujo, la Programación Orientada a Objetos, consta de objetos, y una clase, define o es como la "plantilla" sobre la cual se construyen los tan mentados.

Constructores:

En una clase existe una función miembro muy particular llamada **Constructor**.
Un constructor es una función que debe tener el mismo nombre que la clase y no debe retornar ningún valor, (ni siquiera void), y se encarga de asignarle valores iniciales, (o simplemente inicializar), a los datos miembros.
En el ejemplo descubrirá que allí no hay ningún constructor definido, cuando ocurre esto se dice que existe un constructor por defecto, que inicializa las variables a cero, si son números, a cadena vacía si son cadenas o a falso si son boléanos.
No obstante hubiera sido correcto haber definido un constructor que se encargara de inicializar los datos miembros.

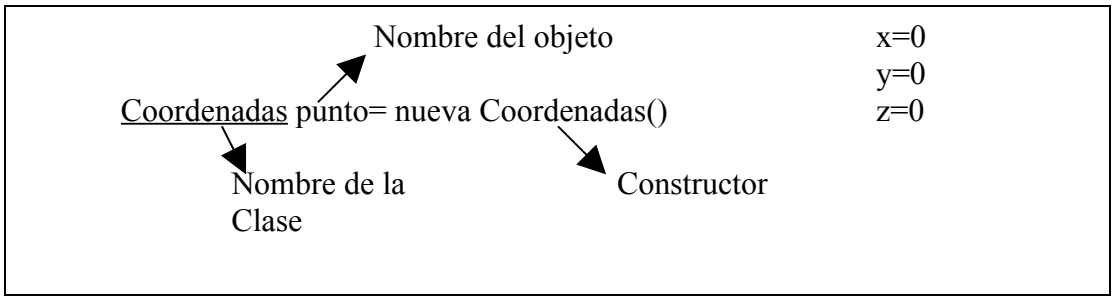
Existen 4 tipos de constructores:

- *Constructor por defecto.*
- *Constructor de Inicialización.*
- *Constructor común.*
- *Constructor de copia.*

El **constructor por defecto** es, como hemos dicho antes, es el que inicaliza los miembros de la clases sino hemos indicado ningún constructor.

```
clase Coordenadas
{
   int x,y,z;
}
```

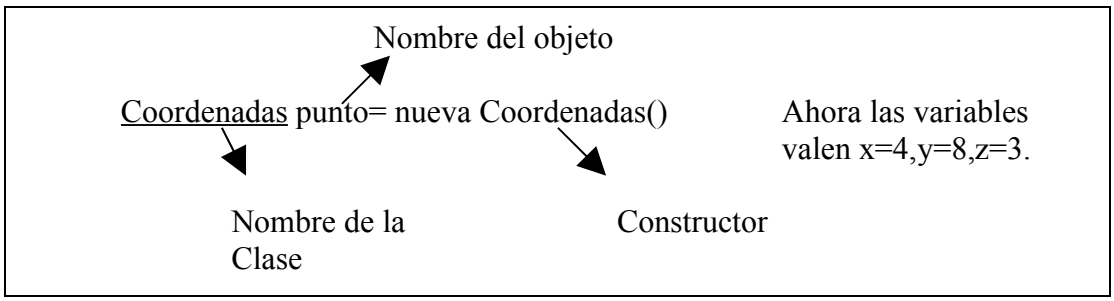
Esta clase no tiene constructor definido por el usuario, pero si existe un constructor por defecto, que inicializa los datos miembros (x,y,z). Para crear un objeto en el programa principal ponemos:



El **Constructor de Inicialización** lo crea el usuario para inicializar los datos miembros a unos valores determinados, sustituye al Constructor por defecto.

```
clase Coordenadas
{
   int x,y,z;

   Coordenadas()
   {
         x=4;
         y=8;
         z=3;
    }
}
```

Para crear un objeto en el programa principal ponemos:



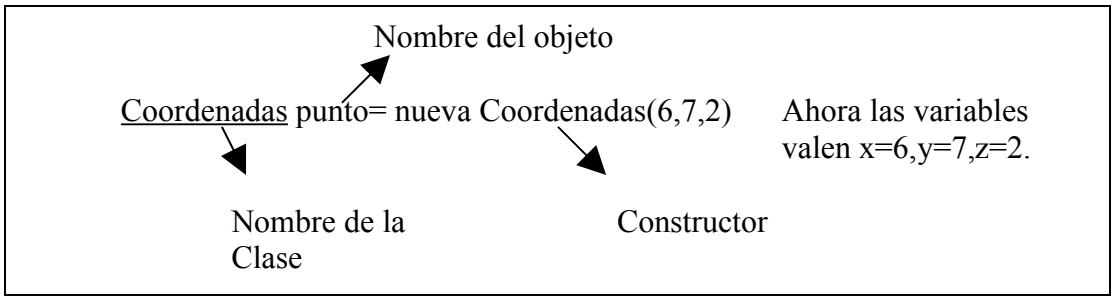
El **constructor común** es aquel que recibe parámetros para asignarles como valores iniciales a los datos miembros, o sea que al crear la instancia, se pasó unos parámetros para inicializar.

```
clase Coordenadas
{
   int x,y,z;

publico:
```

```
   Coordenadas(int p, int q, int t) {x=p; y=q; z=t;}     //Constructor común.

};
```

Cuando se crea el objeto se escribe:

Para crear un objeto en el programa principal ponemos:



El **constructor de copia** se utiliza para inicializar un objeto con otro objeto de la misma clase.

```
clase Coordenadas
{
   int x,y,z;

publico:
    Coordenadas ( int p, int q, int t) {x=p; y=q; z=t;}        //Constructor común.
    Coordenadas(Coordenadas c)    //Constructor de copia.
      {
        x=c.x;
        y=c.y;
        z=c.z;
      }

};
```

Cuando se crea el objeto se escribe:

```
//Creación de un objeto  con lo valores iniciales 6, 7 y 3.
Coordenadas punto1= nueva Coordenadas (6,7,2)

Coordenadas punto2=nueva Coordenadas (punto1)
```

Sobrecarga de funciones:

Habrá advertido en el último ejemplo de la clase, donde se ve el *constructor de copia*, que también se define un *constructor común*. Bueno, eso es posible, una clase puede tener varios constructores, que se irán usando de acuerdo a como uno cree el objeto, (pasándole o no parámetros).

Pero, observe nuevamente, esta vez más detalladamente, la clase..., ¿no encuentra otra cosa extraña?.

Los constructores son funciones, ¿¿¿cómo permite el compilador dos funciones con el mismo nombre???.

El polimorfismo permite funciones con el mismo nombre, el único requisito es que cada una de ellas tenga diferente número y/o tipo de parámetros.
Esta cualidad, que no se aplica solamente a los constructores y funciones miembros de una clase, se llama *Sobrecarga de funciones* o *Polimorfismo*.
Cuando se llama a la función, se selecciona de todas las funciones sobrecargadas aquella que se ajusta de acuerdo con los parámetros pasados, en cantidad y tipo.

Especificadores de acceso:

Ya había dicho que **por defecto** los **datos miembros** de una clase son **privados**. ¿Qué significa esto?.
Que sólo las funciones miembros públicas de la misma clase tienen acceso a ellos. Si lo desea puede escribir la cláusula *privado* al momento de declarar los datos.
En cambio la cláusula *publico* es obligatoria cuando se desea declarar un dato público y este dato estará disponible para cualquier función del programa.
Existe una cláusula más, *protegido*. Los datos definidos a continuación de esta cláusula están restringidos para cualquier función externa a la clase, pero son públicos para la propia clase y los miembros de clases derivadas.

- **Herencia.**
- **Tipos de herencias.**
- **Ejemplo de herencia.**
- **Accesibilidad de miembros en clases derivadas.**

HERENCIA:

Una de las características más importantes de la POO, es la capacidad de derivar clases a partir de las clases existentes, (o sea obtener una nueva clase a partir de otra). Este procedimiento se denomina Herencia, puesto que la nueva clase "hereda" los miembros, (datos y funciones) de sus clases ascendientes y puede anular alguna de las funciones heredadas. La herencia permite reutilizar el código en clases descendientes.
Cuando una clase se hereda de otra clase, la clase original se llama *clase base* y la nueva clase se llama *clase derivada.*

Quizás lo más difícil al escribir programas que utilicen clases, es el diseño de las mismas, lo que involucra una abstracción del objeto que van a representar, pero más difícil aún es diseñar una clase que luego sirva como "base" para nuevas clases derivadas. Aquí además de la abstracción hay que seguir ciertas reglas de accesibilidad, que más adelante de describen.

Veamos un ejemplo:

Uno podría definir una clase Empleado.
Esta clase, básica, solamente tendrá dos datos miembros: el Apellido y el Salario, dos constructores comunes y dos métodos, (o funciones miembros), que nos permitirán obtener el Apellido y el Salario del empleado.
Así podría ser la clase Empleado:

```
clase Empleado
{
Protegido:
   String ape;
   double sueldo;
publico:
   Empleado()
   {
      ape="";
      sueldo=0;
   }
   Empleado(String ap, double s)
   {
     ape = ap;
      sueldo=s;
   }
```

```
   String ObtenerApellido()
   {
        return ape;
    }
   double ObtenerSueldo()
   {
        return sueldo;
   }

};
```

Un programa que cree una instancia de esta clase, deberá usar alguno de los dos constructores, se puede aprovechar el constructor que recibe como parámetros los valores iniciales para el Apellido y el Salario, ya que el otro inicializa con cadena vacía y 0, no es muy útil.

Luego si uno quiere saber cual es el apellido del empleado deberá usar la función ObtenerApellido() ya que los datos miembros son protegidos y "no se ven" en el programa, (esto es "encapsulamiento"). Ofrecemos funciones para acceder a los datos miembros, a modo de protección de la información.
Así que, como ejemplo se podría escribir, dentro de la función main(), lo siguiente:

```
Empleado e=nuevo Empleado(“Perez”,1000)
Escribir(“Apellido:”,e.ObtenerApellido())
Escribir(“Sueldo:”,e.ObtenerSalario())
```

Hasta aquí una clase sencilla y hasta muy poco útil diría, pero nos va a servir para ejemplificar un caso sencillo de herencia.

Existen dos tipos de Herencia:

Simple: Una clase se deriva de sólo una clase base.
Múltiple: Una clase se deriva de más de una clase base.

Nuestro ejemplo será de herencia simple.

Ante todo una pregunta: ¿qué representa nuestra clase Empleado?.

Bueno, la definimos con la intensión que represente un empleado cualquiera. Empleados existen muchos, sea cual sea la empresa en la que trabajan, todos tienen actividades específicas, por ejemplo un obrero, un jefe de sector, un gerente, etc. Empleado es una clase, (por como la diseñamos, aunque sea muy simple), que no hace diferencias entre los empleados, es, digamos, una clase general.
Ahora bien, por lo general, un gerente, (que como dijimos, es un empleado), tiene atributos particulares que lo hace diferente a otro empleado, por ejemplo tiene a su cargo un departamento, tiene una secretaria a su disposición, etc. Por lo tanto tenemos un pequeño problema con nuestra clase Empleado para poder representar un gerente,

puesto que se limita solamente al Apellido y el Salario, nos faltarían datos miembros para el departamento que tiene a su cargo y la secretaria.
Podríamos definir una nueva clase Gerente con los datos miembros Apellido, Salario, Dpto, Secretaria y las funciones miembros ObtenerApellido(), ObtenerSalario(), ObtenerDpto() y ObtenerSecretaria(). Sí y no estaría mal, pero sería redundante, pues podríamos derivar la nueva clase Gerente de Empleado, puesto que los datos y funciones miembros nos resultan útiles.
Derivemos, entonces, la clase Gerente de Empleado:

```
//Definición de la clase Gerente heredada de Empleado
//*****************************************************************
clase Gerente: extendida de Empleado
{
   String dpto;
   String secretaria;

publico:
   Gerente(String n, double s, String d, String sec)
   {
      ape=n;
      sueldo=s;
      spot=d;
      secretaria=sec;
   }
   String ObtenerSecretaria()
   {
        return secretaria;
    }

   String  ObtenerDpto()
   {
        return dpto;
    }
};
```

La definición de clase Gerente**:** extendida de Empleado indica que Gerente heredará de Empleado, todos los datos y funciones miembros.
Una clase hereda de otra todos los datos y funciones miembros que **no sean privados**.

En la función main se podría escribir, para probar la funcionalidad de nuestra nueva clase derivada, lo siguiente:

```
   Gerente g=nuevo Gerente("Perez", 2500.60, "Sistemas", "Juana");
   Escribir(“Apellido:”,g.ObtenerApellido())
   Escribir(“Sueldo:”,g.ObtenerSalario());
   Escribir(“Secretaria:”,g.ObtenerSecretaria())
   Escribir(“Departamento:”,g.ObtenerDpto())
```

Creamos una instancia de Gerente y probamos los métodos de la clase.
Hay que destacar que cuando definimos la clase Gerente no escribimos nada con respecto a los atributos Apellido y Salario y los métodos ObtenerApellido() y ObtenerSalario(), y sin embargo los podemos usar. Esto es la herencia. La posibilidad de poder reutilizar código.
En la clase derivada sólo hay que definir los datos y funciones miembros exclusivos de esa clase, (en nuestro ejemplo el nombre del Dpto y la secretaria y las funciones que permiten obtener estos valores).