PROGRAMACIÓN
1º Desarrollo de Aplicaciones Web

Introducción a la Programación Orientada a Objetos

Rafael del Castillo Gomariz
IES Gran Capitán

Programación orientada a objetos.

Índice.

1. Introducción.

La POO es un paradigma de programación, no es un lenguaje específico ni una tecnología, que se ha constituido en una de las formas de programar más populares y muchos de los lenguajes que usamos hoy día lo soportan o están diseñados bajo ese modelo.

Lo que caracteriza a la POO es que intenta llevar al mundo del código lo mismo que encontramos en *El Mundo Real elevando* el nivel de abstracción de la programación estructurada. Cuando miramos a nuestro alrededor ¿qué vemos? pues, cosas, objetos, pero podemos reconocer estos objetos porque cada objeto pertenece a una clase, eso nos permite distinguir, por ejemplo, un perro de un auto (porque son de clases diferentes) y también un TV de otro (porque, aunque sean iguales, cada uno es un objeto distinto). Éste es el modelo que la POO intenta seguir para estructurar un sistema.

La POO difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

1.1. Historia.

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones. En este centro, se trabajaba en simulaciones de naves, que fueron confundidas por la explosión combinatoria de cómo las diversas cualidades de diferentes naves podían afectar unas a las otras. La idea surgió al agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus propios datos y comportamientos. Fueron refinados más tarde en Smalltalk, desarrollado en Simula en Xerox PARC (cuya primera versión fue escrita sobre Basic) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "sobre la marcha" (en tiempo de ejecución) en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos se fue convirtiendo en el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las *Interfaces Gráficas de Usuario*, para las cuales la programación orientada a objetos está particularmente bien adaptada. En este caso, se habla también de programación dirigida por eventos.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, Visual Basic, Lisp, Pascal, entre otros. También se construyeron nuevos lenguajes de programación en los las características de la programación orientada a objetos eran parte fundamental de su arquitectura, como Java o Python por ejemplo.

1.2. Objetos

Los objetos son entidades que tienen un determinado estado, comportamiento (método) e identidad:

- El estado está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).

- El comportamiento está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La identidad es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

1.3. Características de la POO

Las características más importantes que solemos encontrar en la gran mayoría de los lenguajes mediante los que podemos realizar POO son las siguientes:

- Abstracción: Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.
- Encapsulamiento: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- Principio de ocultación: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.
- Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento

correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

- Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

2. Declaración y definición de clases.

2.1. Concepto de clase.

Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común. Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (instanciación) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.

Las clases presentan el estado de los objetos a los que representan mediante variables denominadas ***atributos***. Cuando se instancia un objeto el compilador crea en la memoria dinámica un espacio para tantas variables como atributos tenga la clase a la que pertenece el objeto.

Los ***métodos*** son las funciones mediante las que las clases representan el comportamiento de los objetos. En dichos métodos se modifican los valores de los atributos del objeto, y representan las capacidades del objeto.

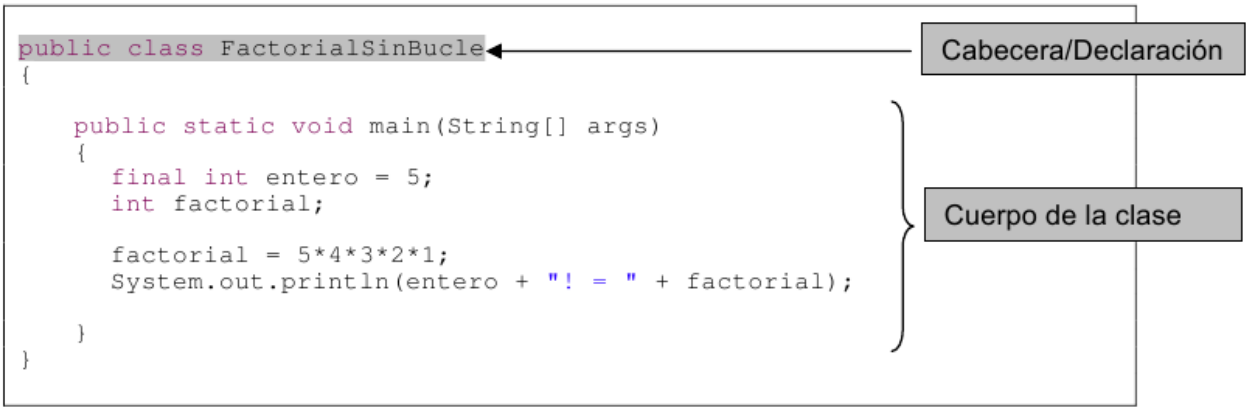
Desde el punto de vista de la programación estructurada, una clase se asemejaría a un módulo, los atributos a las variables globales de dicho módulo, y los métodos a las funciones del módulo.

2.2. Definición de clase.

La sintaxis de la definición de una clase consta de dos partes claramente diferenciadas: su cabecera y el cuerpo de la clase.

La cabecera nos dará cuenta de los distintos aspectos que deberán ser tenidos en cuenta para el manejo de esta clase, es decir, nos aporta información fundamental sobre la clase en sí y constituye de alguna manera la declaración de esa clase.

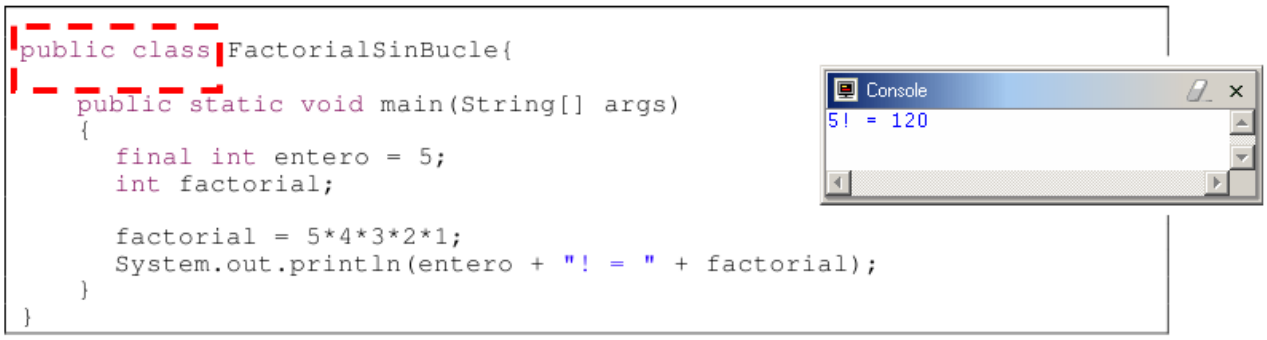
Mientras que el cuerpo de la clase se reserva para la declaración de los atributos y métodos que serán capaces de ejecutar los objetos generados a partir de esa clase.



2.3. Cabecera de una clase

En la declaración de una clase, se especifica el identificativo de la clase, y una serie de características propias de los Lenguajes Orientados a Objetos, que puede tener una clase. Vamos a usar Java como base para explicar como sería la sintaxis general de la declaración de una clase, ya que en el resto de lenguajes de POO es muy similar:

[modificador] class NombreClase [extends NombreSuperclase] [implements listaDeInterfaces]



- *Superclase*: clase de la que deriva (herencia), si no se indica significa que la clase no desciende "explícitamente" de ninguna otra (por ejemplo, en el caso de Java todas las clases son descendientes, directa o indirectamente, de la clase Object que es la raíz de toda la jerarquía de clases en Java).
- Lista de *interfaces*: Un interface es un conjunto de constantes (no variables) y declaración de métodos, únicamente su declaración (lo que en C equivaldría al prototipo), no su implementación o cuerpo.
- Los *modificadores de clase* son palabras reservadas que se anteponen a la declaración de clase y que especifican la forma en la que es accesible la clase, para otras clases. Los modificadores posibles son los siguientes:
  - *public*: todas las clases pueden acceder a esta clase.
  - *abstract*: Sirve para definir clases abstractas que no pueden ser instanciadas, sólo podrán usarse como superclases y normalmente no tienen codificados todos sus métodos.
  - *final*: Este modificador sirve para impedir que una clase tenga clases derivadas (no pueden heredar de ella).

2.4. Cuerpo de la clase.

El cuerpo de la clase sigue a la declaración de la clase y en el caso de Java está contenido entre la pareja de llaves ({ y }). El cuerpo de la clase contiene las declaraciones de las variables o atributos

de la clase, y también la declaración y la implementación de los métodos que operan sobre dichas variables.

- Declaración de variables de instancia: el estado de un objeto está representado por sus atributos (variables de instancia). Las variables de instancia se declaran dentro del cuerpo de la clase. Típicamente, las variables de instancia se declaran antes de la declaración de los métodos. Estas variables de instancia pueden ser públicas o privadas (no se puede acceder directamente a las mismas). Es aconsejable poner los atributos privados y acceder a los mismos a través de métodos específicos.
- Implementación de métodos: los métodos de una clase determinan los mensajes que un objeto puede recibir. Las partes fundamentales de un método son el valor de retorno, el nombre, los argumentos (opcionales) y su cuerpo. La sintaxis de un método (en Java) es la siguiente:

  <otrosModificadores> valorRetorno nombreMetodo(<lista de argumentos>)
  {
  / Cuerpo del método */*
  sentencias;
  }

Cuando se llama a un método de un objeto se dice comúnmente que se envia un mensaje al objeto.

Ejemplo:

```
/* Usuario.java */

class Usuario
{
    String nombre;
    int edad;
    String direccion;

    void setNombre(String n)
    {
      nombre = n;
    }

    String getNombre()
    {
      return nombre;
    }

    void setEdad(int e)
    {
      edad = e;
    }

    int getEdad()
    {
      return edad;
    }
```

```
    void setDireccion(String d)
    {
      direccion = d;
    }

    String getDireccion()
    {
      return direccion;
    }
}
```

3. Constructores y destructores.

3.1. Constructores.

El objetivo del *constructor* es el de inicializar un objeto cuando éste es creado. Asignaremos los valores iniciales así como los procesos que ésta clase deba realizar.

Se utiliza para crear tablas de métodos virtuales y poder así desarrollar el polimorfismo. Al utilizar un constructor, el compilador determina cual de los métodos va a responder al mensaje (virtual) que hemos creado. Tiene un tipo de acceso, un nombre y un paréntesis.

Un *constructor* suele ser un método especial dentro de una clase, que se llama automáticamente cada vez que se crea un objeto de esa clase.

En el caso de Java posee el mismo nombre de la clase a la cual pertenece y no puede regresar ningún valor (ni siquiera se puede especificar la palabra reservada void). Cuando en una clase no se escribe propiamente un constructor, java asume uno por defecto.

Es habitual crear varios constructores para especificar como debe crearse el objeto ante diferentes situaciones (si no pasamos parámetros al crear el objeto, si le pasamos parámetros de uno u otro tipo, etc...).

Ejemplo:

```
Fraccion c1 = new Fraccion();
Fraccion c2 = new Fraccion(5,4);
```

c1 es un objeto de clase *Fraccion* sin especificar numerador y denominador, pero c2 es otro de la misma clase donde si especificamos numerador y denominador.

3.2. Destructores.

Un destructor es un método especial dentro de una clase que es invocado cuando el objeto deja de utilizarse, al alcanzar el flujo del programa el fin del ámbito en el que está declarado el objeto. No es necesario llamarlo explícitamente ya que se invoca de forma automática. Sus principales cometidos son:

- Liberar los recursos computacionales que el objeto de dicha clase haya adquirido en su tiempo de ejecución al expirar éste.
- Quitar los vínculos que pudiesen tener otros recursos u objetos con éste.

En el caso de C++ el destructor se nombra con el mismo nombre de la clase precedido de un símbolo ~. Ejemplo

```
#include <iostream>
#include <string>

class foo
{
	public:

	foo( void )
	{
		print( "foo()" );
	}

	~foo( void )
	{
		print( "~foo()" );
	}

	void print( std::string const& text )
	{
		std::cout << static_cast< void* >( this ) << " : " << text << std::endl;
	}


int main( void )
{
	foo array[ 3 ];
/*
	Cuanto la 'main' termina, el destructor es invocado para cada elemento de 'array'.
	La primera instancia creada es la última en ser destruída.
*/
}
```

4. Interfaces.

Un/a *interface* es un conjunto de constantes (no variables) y declaración de métodos, únicamente su declaración, no su implementación o cuerpo (en el caso de Java esta posibilidad corrige el inconveniente de que no pueden declararse clases descendientes de más de una superclase -herencia múltiple-).

El aspecto de un *interface* (en Java) puede ser algo similar a:

```
public interface OperacionesAritmeticas {

	static final int AtributoConstante;
	//... más declaraciones de constantes

	Object Sumar (Object a, Object b);
	Object Restar (Object a, Object b);

	//... más declaraciones de métodos SIN IMPLEMENTAR
}
```

En lo referente a la declaración de una clase es necesario dar cuenta explícita de todos aquellos interfaces de los que se va a hacer uso dentro de una clase.

En el caso de Java para indicar que una clase va a implementar una serie de interfaces, se usa la palabra reservada *implements*, seguida de la lista de interfaces que no es otra cosa que simplemente los nombres de los interfaces que se quieran especificar separados entre sí por medio de comas. Por ejemplo:

class Nif extends Dni implements OperacionesAritméticas,OperacionesLógicas

En la interface *OperacionesAritmética*s pueden estar declarados, por ejemplo, los métodos *suma()*, *resta()*, etc. Mediante la declaración de clase del ejemplo, se obliga a que en la clase *Nif* haya que implementar, dotar de cuerpo, a todos los métodos de la interfaz, de lo contrario el compilador mostrará el correspondiente mensaje de error. En este caso, también deberán redefinirse todos los métodos de la interfaz *OperacionesLógicas*.

En la interface se especifica **qué se debe hacer** pero no su implementación. Serán las clases que implementen estas interfaces las que describan la logica del comportamiento de los métodos.

El uso de interfaces proporciona las siguientes ventajas:

- Organizar la programación.
- Obligar a que ciertas clases utilicen los mismos métodos (nombres y parámetros).
- Establecer relaciones entre clases que no estén relacionadas.

5. Encapsulación.

Las atributos del objeto se localizan en el núcleo del objeto, los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa. Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama encapsulamiento. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

El encapsulamiento de aributos y métodos en un componente de software ordenado es una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

- Modularidad, esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.
- Ocultamiento de la información, es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependan de ello.

Los objetos proveen el beneficio de la modularidad y el ocultamiento de la información. Las clases proveen el beneficio de la reutilización. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos. La realidad es diferente: los atributos se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor asociado, para cada variable, diferente al que tienen

para esa misma variable los demás objetos. Los métodos, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código. De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

La encapsulación da lugar a que las clases se dividan en dos partes:

- Interfaz: Captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
- Implementación: Comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado.

6. Herencia.

La *herencia* es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la POO. Esto es, el código de cualquiera de las clases puede ser utilizado sin más que crear una clase derivada de ella, o bien una subclase.

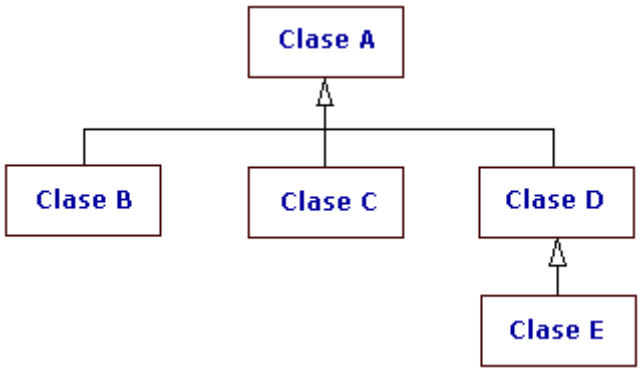
Hay dos tipos de herencia: Herencia Simple y Herencia Múltiple. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java sólo permite herencia simple.

6.1. Superclase y Subclases

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la POO todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene sólo una clase padre. La clase padre de cualquier clase es conocida como su superclase. La clase hija de una superclase es llamada una subclase.

Una superclase puede tener cualquier número de subclases, mientras que una subclase puede tener sólo una superclase. Ejemplo:



- A es la superclase de B, C y D.
- D es la superclase de E.
- B, C y D son subclases de A.
- E es una subclase de D.

6.2. Herencia y ocultación de información

El diseñador puede definir qué variables de instancia y métodos de los objetos de una clase son visibles. En C++ y Java esto se consigue con las especificaciones *private, protected y public*. Sólo las variables y métodos definidos como públicos en un objeto serán visibles por todos los objetos.

En cuanto a las subclases, que heredan las estructuras de las superclases, el diseñador puede controlar qué miembros de las superclases son visibles en las subclases. En el caso de Java y C++ los especificadores de acceso (*private, protected, public*) de los miembros de la superclase afectan también a la herencia:

- Private: ningún miembro privado de la superclase es visible en la subclase.
- Protected: los miembros protegidos de la superclase son visibles en la subclase, pero no visibles para el exterior.
- Public: los miembros públicos de la superclase siguen siendo públicos en la subclase.

6.3. Redefinición de métodos

En la clase derivada se puede redefinir algún método ya definido en la clase base. Para redefinir un método en la subclase, basta con declarar una función miembro con el mismo nombre. Si en una clase en particular se invoca a un método, y el método no está definido en la misma, es buscado automáticamente en las clases superiores. Sin embargo, si existieran dos métodos con el mismo nombre y distinto código, uno en la clase y otro en una superclase, se ejecutaría el de la clase, no el de la superclase.

Por lo general, siempre se puede acceder explícitamente al método de la clase superior mediante una sintaxis diferente, la cual dependerá del lenguaje de programación empleado.

6.4. Ventajas

- Ayuda a los programadores a ahorrar código y tiempo, ya que la clase padre ha sido implementada y verificada con anterioridad, restando solo referenciar desde la clase derivada a la clase base.
- Los objetos pueden ser construidos a partir de otros similares. Para ello es necesario que exista una clase base y una jerarquía de clases.
- La clase derivada puede heredar código y datos de la clase base, añadiendo código o modificando lo heredado.
- Las clases que heredan propiedades de otra clase pueden servir como clase base de otras.

7. Polimorfismo y sobre-escritura.

El polimorfismo se aplica en los métodos y se refiere a que el mismo método puede ser usado para diferentes fines según se necesite.

Para explicar mejor este concepto vamos a echar mano de la implementación en java. En ese lenguaje se tienen dos clases de polimorfismo uno se llama sobrecarga y el otro se llama sobre escritura. Esta definición puede variar según la implementación, sin embargo el concepto de polimorfismo es el mismo.

7.1. Polimorfismo de Sobrecarga

La sobrecarga se refiere a que un método puede implementarse para varios fines con el mismo nombre en la misma clase, el ejemplo más simple se observa con los métodos constructores que se pueden implementar con varios parámetros diferentes como el constructor vacío y el constructor lleno.

Ejemplo:

```
//Ejemplo de polimorfismo de sobrecarga en Java

public class Celular{
   int numero;

   //dos metodos constructor
   public Celular(){
   }

   public Celular(String numero){
   }
}
```

7.2. Polimorfismo de sobre-escritura

La sobre-escritura se aplica cuando una clase hereda de otra u otras, en ese sentido la clase hija hereda sus atributos y métodos, sin embargo esto no implica que los métodos se tengan que utilizar estrictamente en la forma que lo hace su padre, en caso de haber cambios en la implementación de un método este se escribe de nuevo en la clase hija con sus particularidades.

Ejemplo:

```
public class Alumno extends Persona {
        protected int matricula;

        public Alumno() {
                super();
                matricula = 0;
        }

        ….
```

8. Clases y métodos abstractos y finales. Genéricos / plantillas.

8.1. Clases y métodos abstractos.

Los programadores pueden implementar superclases llamadas clases abstractas que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente, la conducta pero gran parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

Una clase abstracta es una que está diseñada sólo como clase base de las cuales se deben derivar clases hijas. Una clase abstracta se usa para representar aquellas entidades o métodos que después se implementarán en las clases derivadas, por ello, no es posible instanciar una clase abstracta, pero sí una clase concreta que implemente los métodos definidos en ella.

Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código). Una clase abstracta tiene al menos un método abstracto. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

Ejemplo en Java:

```
abstract class FiguraGeometrica {
   . . .
   abstract void dibujar();
   . . .
}

class Circulo extends FiguraGeometrica {
   . . .
   void dibujar() {
      // codigo para dibujar Circulo
      . . .
   }
}
```

En C++ los métodos de las clases abstractas se definen como funciones virtuales puras. Ejemplo:

```
class Abstracta
{
 public:
   virtual int metodo() = 0;
};

class ConcretaA : public Abstracta
{
 public:
    int metodo()
    {
     //haz algo
     return foo () + 2;
    }
};
```

```
class ConcretaB : public Abstracta
{
  public:
     int metodo()
     {
        //otra implementación
        return baz () - 5;
     }
};
```

En el ejemplo, la clase *ConcretaA* es una implementación de la clase *Abstracta*, y la clase *ConcretaB* es otra implementación. Debe notarse que el = 0 es la notación que emplea C++ para definir funciones virtuales puras.

8.2. Plantillas.

Con las *plantillas* podemos definir clases y funciones cuyas variables no están definidas en tiempo de diseño, sino que se definen en tiempo de compilación. Veamos un ejemplo de función genérica para intercambiar valores en C++:

```
#include <iostream>
using namespace std;

template<class Type>
void intercambio(Type &x,Type & y)
{
   Type aux=x;
   x=y;
   y=aux;
}

int main()
{
   int ia=10,ib=15;
   intercambio(ia,ib);
   cout<<ia<<" "<<ib<<endl;
   float fa=11.1,fb=10.1;
   intercambio(fa,fb);
   cout<<fa<<" "<<fb<<endl;
   return 0;
}
```

template<class Type> es la forma de indicar al compilador de C++ que vamos a generar una clase template. Además, le estamos diciendo que vamos a definir un tipo genérico llamado Type. El elemento Type, es una variable genérica que no se define en tiempo de diseño, sino que se define en tiempo de compilación, esto significa que antes de llamar a la función no sabemos qué será Type, podra ser un int, double, float o lo que sea, pero aun no está definido.

8.3. Clases genéricas.

El concepto de plantilla se extiende a las clases. Imaginemonos que tenemos creadas distintas clases

para los distintos números Integer, Float y Double, hemos tenido que reimplementar la misma funcionalidad hasta tres veces haciendo cambios mínimos. Con C++, es posible evitar esto definiendo una plantilla. Vamos a crear la clase Number para demostrar esto. Creamos number.h

```
#ifndef _Number_H_
#define _Number_H_

namespace values
{
/**\brief This class represents a generic number
*/
template<class Type>
class  Number
{
public:
   /**Empty constructor
   */
   Number()  {_value=0;}
   /**Parametrized constructor
   */
   Number(Type v)  {_value=v;}
   /**Sets the current value
   */
   void setValue(Type v){ _value=v;}
   /**Returns the current value
   */
   Type getValue(void){ return _value;}
private:
   Type _value;
};
};
#endif
```

Obsérvese la sintaxis. Lo primero es el uso de la palabra clave template<class Type>. Esto es para decir que estamos creando una función genérica. Pero después, vemos que hemos incluido el código directamente en la declaración.

Ahora, el main.cpp

```
#include <iostream>
#include <number.h>
using namespace std;
using namespace values;

int main()
{
Number<int> MyInteger(10);
Number<float> MyFloat(11.1);
Number<double> MyDouble(111.1);
Number<long int> MyLInt(1111);
```

```
Number<unsigned int> MyUInt(1111);
}
```

Al declarar en main.cpp Number<int> MyInteger(10) creamos un objeto de la clase Number, donde Type se sustituye por int. Más abajo, lo sustituimos por float. Y así tanto como nos de la gana. El caso es que para MyInteger, Type es int pero par MyFloat es float. Es muy útil.

En el caso de C++ todo el código de la clase que usa plantillas debe ir en el fichero de cabecera (.h). Esto se debe a que la clase no se compila hasta que no se instancia un objeto de ella.

Bibliografía:

http://thefricky.wordpress.com/poo/
http://es.wikipedia.org/wiki/C%2B%2B
http://ucoedi.wikispaces.com/
http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos