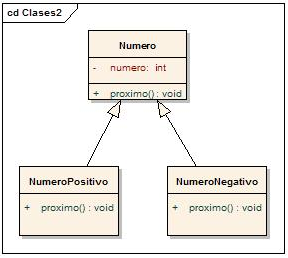
**ASPECTOS AVANZADOS DE OOP**

**Polimorfismo**

**Definición**

El polimorfismo se produce cuando un método adopta más de una forma. Un método puede modificar su comportamiento de acuerdo a su necesidad.

Como ejemplo se puede utilizar lo siguiente:



   public class Numero{

        protected int numero;

        public void proximo(){

        }

        public String toString(){

          return String.valueOf(numero);

        }

  }

  public class NumeroPositivo extends Numero {

    public NumeroPositivo(){

      numero = 0;

    }

    public void proximo(){

      numero = numero + 1;

    }

  }

  public class NumeroNegativo extends Numero {

    public NumeroNegativo(){

      numero = -1;

    }

    public void proximo(){

      numero = numero - 1;

    }

  }

  public class Programa {

public static void main(String[] args){

      Numero positivo = new NumeroPositivo();

      Numero negativo = new NumeroNegativo();

      System.out.println("numero positivo inicial:"+positivo);

      System.out.println("numero negativo inicial:"+negativo);

      positivo.proximo();

      negativo.proximo();

      System.out.println("numero positivo proximo:"+positivo);

      System.out.println("numero negativo proximo:"+negativo);

      positivo.proximo();

      negativo.proximo();

      System.out.println("numero positivo proximo:"+positivo);

      System.out.println("numero negativo proximo:"+negativo);

    }

  }

El resultado de este programa es el siguiente:

* número positivo inicial:0
* número negativo inicial: -1
* número positivo próximo: 1
* número negativo próximo: -2
* número positivo próximo: 2
* numero negativo próximo: -3

La conclusión que debemos observar en este ejemplo es que se tienen dos instancias de Número, una instancia de NumeroPositivo y una de NumeroNegativo, cada cual tiene un comportamiento distinto. Sin embargo a la hora de declararlo se los declaró en su forma genérica de Número y se tiene el método proximo() asociado.

El polimorfismo ha permitido que cada uno se comporte de manera distinta, y sin embargo parece llamarse siempre al mismo método. Es decir, es distinto el próximo negativo del próximo positivo, y eso se refleja en el resultado.

Otra cosa que puede verse en este ejemplo es que al imprimir los objetos positivo y negativo se llama automáticamente al método toString(), es el modo en el que java imprime sus objetos en pantalla. Y el aprovechamiento del mecanismo de herencia, en la programación del método toString() que retorna el valor de número como un String, es decir, algo que puede imprimirse fácilmente. Al llamarlo en un NumeroPositivo, como este no lo tiene programado, se ejecuta el que está definido en la clase padre Número. Al llamarlo en un NumeroNegativo también se llama al de la clase padre.

**Con redefinición**

Se llama polimorfismo con redefinición cuando en una clase sobreescribimos un método, definido en la superclase.

Ejemplo en Java:

public class Persona{

public void cantar(){

          // Esto es un comentario. Aqui va la definicion.

    }

  }

  public class Cantante extend Persona{

        public void cantar(){

      // Esto es un comentario. Aqui va la redefinicion.

}

 }

En este caso se puede ver que todas las personas pueden cantar(), sin embargo no es lo mismo cantar() en la clase Cantante, pues el cantante logra mayor escala, mayor volumen de voz, y entona de una forma muy preparada. Un cantante logra llegar más lejos con su canto que una persona común. Al redefinir el método en la clase Cantante se logra que se ejecute el método cantar() de cantante en lugar del método cantar() de persona. Por lo tanto, la especialización que se logra con este mecanismo permite cambiar el comportamiento de una familia de clases.

**Sin redefinición**

Se llama polimorfismo sin redefinición cuando un método tiene distinto comportamiento de acuerdo a la clase a la que pertenece.

Ejemplo en Java:

public class Persona{

        public void beber(Agua unPocoDeAgua){

          // Esto es un comentario. Aca va una definicion.

    }

  }

   public class Animal{

        public void beber(Agua unPocoDeAgua){

          // Esto es un comentario. Aca va otra definicion.

    }

  }

En este caso ambas clases logran beber(), es decir que ambas clases tienen definido el método beber, solo que uno de los métodos beber lo hace como humano, y el otro lo hace como un animal, el comportamiento es bien distinto, aunque el nombre sea el mismo.

**Clase Abstracta**

**Definición**

Una clase abstracta representa un concepto abstracto que no debe ser instanciado. Es simplemente una clase que no se puede instanciar. Sirve como base para que otras clases hereden de ella sus atributos y métodos, pero no tiene sentido por sí sola como instancia. Los métodos abstractos no pueden estar definidos en la clase.

Pero deberán ser obligatoriamente definidos en las subclases.

**Definición**

Ejemplo en Java:

  public abstract class Persona{

        //Atributos

        private String nombre;

        private Date fechaDeNacimiento;

    //Metodos

    public abstract void estudiar();

        public String getNombre(){

          return this.nombre;

        }

  }

U**tilización**

 public class Hombre extends Persona{

        // Metodo que se implementa Obligatoriamente,

    // por estar definido como abstracto en la clase padre

    public void estudiar(){

      // Aqui va el codigo de estudiar. Leer, entender, reflexionar.

    }

  }

**Interfaz**

**Qué es una interfaz**

Es una declaración de comportamiento, es decir un conjunto de métodos sin su implementación. Define un protocolo de comportamiento. Es un contrato que publica una clase para ser utilizada por otras clases. Puede heredar de otras interfases.

**Definición**

Ejemplo en Java, definición:

public interface Imprimible{

        public void imprimir();

        public void imprimir(Tamano t, Estilo s);

  }

**Utilización**

Ejemplo en Java, utilización

public class Documento implements Imprimible{

        public void imprimir(){

          // Esto es un comentario. Aqui va la definicion.

   }

     public void imprimir(Tamano t, Estilo s){

      // Esto es un comentario. Aqui va la definicion.

    }

}

Un ejemplo de utilización de una interfaz, puede ser el siguiente, se tiene un Aeropuerto en el que se puede dar permiso de aterrizar.

 public class Aeropuerto{

     public void darPermisoDeAterrizar(Volador v){

      // Aqui va el código de despejar pista y permitir aterrizar al Volador v.

      // En el programa principal se puede iterar sobre una lista de Voladores e ir dando

      // permiso de aterrizaje a cada volador, sin especificar qué tipo especifico

      // de volador se trata

      v.aterrizar();

     if( v instanceof AvionPrivado ){

        AvionPrivado ap = (AvionPrivado)v;

        System.out.println(“Aterrizo Licencia: ”+ap.getLicencia());

      }

      if( v instanceof AvionDePasajeros ){

        AvionDePasajeros ap = (AvionDePasajeros)v;

        System.out.println(“Aterrizo Aerolinea: ”+ap.getAerolinea());

      }

          if( v instanceof Superman ){

            Superman ap = (Superman)v;

            System.out.println(“Aterrizo Nombre: ”+ap.getNombre());

      }

    }

  }

Luego tenemos distintos TransportesAereos, un AvionDeLinea, y un AvionPrivado, y luego tenemos a Superman.

Entonces lo que se hace es utilizar la interfaz para definir un mismo comportamiento a clases bien distintas, porque no heredan de la misma clase genérica:

public interface Volador{

    public void aterrizar(); // Obligamos a cada Volador a programar su modo de aterrizar.

  }

  public abstract class TransporteAereo{

  }

  public class AvionPrivado extends TransporteAereo implements Volador{

    public void aterrizar(){

      // Aqui va el codigo de modo que tiene de aterrizar un avion privado

    }

   private String licencia;

   public String getLicencia(){

          return licencia;

        }

        public void setLicencia(String lic){

          this.licencia = lic;

    }

  }

public class AvionDePasajeros extends TransporteAereo implements Volador{

    public void aterrizar(){

      // Aqui va el codigo de modo que tiene de aterrizar un avion de pasajeros

    }

    private String aerolinea;

        public String getAerolinea(){

          return aerolinea;

        }

        public void setLicencia(String ae){

          this.aerolinea = ae;

        }

  }

public class Superman extends Superhereo implements Volador{

        public void aterrizar(){

      // Aqui va el codigo de modo que tiene de aterrizar de Superman

    }

        public String getNombre(){

          return “Kent, Clark”;

        }

  }

public class Programa{

        public Programa(){

        }

        public static void main(String[] args){

          // Una inicializacion que puede existir en cualquier lado del programa

     AvionPrivado v1 = new AvionPrivado();

      v1.setLicencia(“L123456F”);

     AvionDePasajeros v2 = new AvionDePasajeros();

     v2.setAerolinea(“EducacionIT Lan”);

     Superman v3 = new Superman();

      Aeropuerto aeropuerto1 = new Aeropuerto();

      aeropuerto1. darPermisoDeAterrizar(v1);

      aeropuerto1. darPermisoDeAterrizar(v2);

      aeropuerto1. darPermisoDeAterrizar(v3);

    }

  }

En este caso se ve cómo se puede llamar al método aterrizar() de los objetos Voladores, porque la interfaz Volador tiene un método aterrizar() y al implementar una interfaz se está diciendo que se tendrá ese comportamiento, es decir que se implementan sus métodos.

**Paquetes**

**Qué es un paquete**

Un paquete es un medio de organización de clases, para agrupar clases e interfaces. Se suelen crear a partir de necesidades funcionales. Pueden contener clases, interfaces y más paquetes.

**Definición**

A continuación, se presenta un ejemplo en Java de cómo definir un paquete:

 package mipaquete.misubpaquete;

public class Persona{

  }

**Utilización**

A continuación se presenta un ejemplo en Java de cómo utilizar un paquete desde una clase ubicada en otro paquete import mipaquete.misubpaquete.\*;

La asociación directa es que una clase o una interfaz está definida en un archivo. Todo archivo debe contener una clase pública al menos, y esta debe tener el mismo nombre que el archivo.

Estas clases e interfaces, estos archivos deben vivir dentro de paquetes, que son equivalentes a los directorios, es decir que la clase Persona, que es pública y está en el archivo Persona.java, debe existir dentro del paquete entidades, es decir el archivo debe vivir dentro del directorio entidades.

Ejemplo:

  /entidades/Persona.java

**La keyword final**

**Definición**

Es una palabra reservada de Java. Tiene una semántica distinta según donde se la utilice.

**Aplicable a atributos**

Aplicada en atributos representa a un valor constante.

Ejemplo:

    public final int valorCuota = 100;

Este valor no puede ser modificado en ningún momento de la programación. Es un valor constante que no se modifica.

**Aplicable a métodos**

Aplicada en métodos representa la no modificación de un método en el caso de querer sobreescribirlo en una subclase.

Ejemplo:

public final void moverPieza(){

  // Esto es un comentario. Aqui va la última definicion.

 }

En caso de heredar de esta clase, no se podrá sobreescribir el método moverPieza(), quedara con su comportamiento original y no se puede modificar.

**Aplicable a clases**

Aplicada en clases significa que esa clase no se puede extender o no se puede hacer una subclase, es decir que no se puede generar una clase que herede de ella. Cierra el arbol de herencias y queda como una hoja de este árbol.

Ejemplo:

public final class Hombre{

}

Es decir, que no se puede heredar de la clase Hombre, esta clase deja cerrado el árbol de herencia. Esta técnica puede ser utilizada cuando se entrega un librería o parte de un programa y se quiere que, quien utilice estas clases, no pueda heredar de estas clases para modificar su comportamiento. Es una medida de seguridad al compartir código o publicar programas.

**Accesibilidad**

**Acceso privado, palabra clave private**

Al definir un atributo, método o clase como privado, se le está dando acceso solo desde la clase donde fue definido. Por ejemplo, en el caso de velocidad de un Auto

public class Auto{

        private int velocidad;

        public void acelerar(){

      this.velocidad += 10;

    }

  }

Por lo tanto, el único modo de acceder a modificar la velocidad es a través del método acelerar. Esta propiedad no está accesible desde ningún otro alcance sino el de la clase.

Para poder acceder al atributo velocidad, solo podrán hacerlo los métodos de la misma clase.

**Acceso por defecto, "default" o "package"**

En el caso del acceso por defecto o "default" o "package", que es el acceso cuando no se declara la accesibilidad a un atributo, método o clase, el acceso es dentro del mismo paquete o directorio. Por ejemplo, si tengo dos clases Auto y Camion, y auto tiene un conductor sin definir su accesibilidad, es decir

package vehiculos;

  public class Auto extends Vehiculo{

        Conductor getConductor(){

      // Aqui se retorna el conductor.

    }

  }

 Luego en la clase Camion que está en el mismo package:

  package vehiculos;

  public class Camion extends Vehiculo{

        public void comunicar(Vehiculo v){

      v.getConductor().conversar(); // Se tiene acceso a getConductor por estar ambas clases

      // en el mismo package.

    }

  }

Para poder acceder al metodo getConductor(), podrán hacerlo únicamente las clases ubicadas en el mismo package o directorio.

**Acceso protegido, palabra clave protected**

Al definir un atributo o método como protegido estamos dejándolo disponible para las clases que heredan de esta clase, ejemplo:

public class Vehiculo{

        protected int cantidadRuedas = 4;

  }

 Luego en la clase Auto, tendremos acceso a cantidad de Ruedas

  public class Auto extends Vehiculo{

        public int getCantidadDeRuedas(){

      return this.cantidadRuedas;

    }

  }

Para poder acceder al atributo cantidadRuedas, solo podrán hacerlo las clases que heredan de Vehículo, es decir la familia de clases.

**Acceso público, palabra clave public**

Definiendo un atributo o método o clase como público permite que se tenga acceso a estos desde cualquier alcance, desde todos lados.

Acceder a los atributos públicos, todos podrán hacerlo.

**Resumen de accesibilidad**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Accesibilidad | Acceso en la misma clase | Acceso en el mismo package | Acceso en el árbol de herencia | Accesos desde todas las clases |
| Private | Si | No | No | No |
| Default o package | Si | Si | No | No |
| Protected | Si | Si | Si | No |
| public | Si | Si | Si | Si |