Unidad V

Herencia y polimorfismo.

5.1 Concepto de herencia y polimorfismo.

En programación orientada a objetos, el **polimorfismo** se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

La apariencia del código puede ser muy diferente dependiendo del lenguaje que se utilice, más allá de las obvias diferencias sintácticas.

Por ejemplo, en un lenguaje de programación que cuenta con un sistema de tipos dinámico (en los que las variables pueden contener datos de cualquier tipo u objetos de cualquier clase) comoSmalltalk no se requiere que los objetos que se utilizan de modo polimórfico sean parte de una jerarquía de clases.

5.2 Definición de una clase base.

Vamos a poner un ejemplo del segundo tipo, que simule la utilización de liberías de clases para crear un interfaz gráfico de usuario como Windows 3.1 o Windows 95.

Supongamos que tenemos una clase que describe la conducta de una ventana muy simple, aquella que no dispone de título en la parte superior, por tanto no puede desplazarse, pero si cambiar de tamaño actuando con el ratón en los bordes derecho e inferior.

La clase *Ventana* tendrá los siguientes miembros dato: la posición *x* e *y* de la ventana, de su esquina superior izquierda y las dimensiones de la ventana: *ancho* y *alto*.

```
public class Ventana {
   protected int x;
   protected int y;
   protected int ancho;
   protected int alto;
   public Ventana(int x, int y, int ancho, int alto) {
      this.x=x;
      this.y=y;
      this.ancho=ancho;
      this.alto=alto;
   }
```

```
//...
}
```

Las funciones miembros, además del constructor serán las siguientes: la función *mostrar* que simula una ventana en un entorno gráfico, aquí solamente nos muestra la posición y las dimensiones de la ventana.

```
    public void mostrar(){
        System.out.println("posición    : x="+x+", y="+y);
        System.out.println("dimensiones : w="+ancho+", h="+alto);
    }
```

La función *cambiarDimension*es que simula el cambio en la anchura y altura de la ventana.

```
    public void cambiarDimensiones(int dw, int dh){
        ancho+=dw;
        alto+=dh;
    }
```

El código completo de la clase base *Ventana*, es el siguiente

```
package ventana;

public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;
    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
    public void mostrar(){
        System.out.println("posición    : x="+x+", y="+y);
        System.out.println("dimensiones : w="+ancho+", h="+alto);
    }
    public void cambiarDimensiones(int dw, int dh){
        ancho+=dw;
        alto+=dh;
    }
}
```

Objetos de la clase base

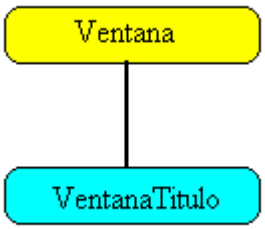
Como vemos en el código, el constructor de la clase base inicializa los cuatro miembros dato. Llamamos al constructor creando un objeto de la clase *Ventana*

```
Ventana ventana=new Ventana(0, 0, 20, 30);
```

Desde el objeto *ventana* podemos llamar a las funciones miembro públicas

```
ventana.mostrar();
ventana.cambiarDimensiones(10, 10);
ventana.mostrar();
```

La clase derivada



Incrementamos la funcionalidad de la clase *Ventana* definiendo una clase derivada denominada *VentanaTitulo*. Los objetos de dicha clase tendrán todas las características de los objetos de la clase base, pero además tendrán un título, y se podran desplazar (se simula el desplazamiento de una ventana con el ratón).

La clase derivada heredará los miembros dato de la clase base y las funciones miembro, y tendrá un miembro dato más, el título de la ventana.

```
public class VentanaTitulo extends Ventana{
   protected String titulo;
   public VentanaTitulo(int x, int y, int w, int h, String nombre) {
      super(x, y, w, h);
      titulo=nombre;
   }
```

extends es la palabra reservada que indica que la clase *VentanaTitulo* deriva, o es una subclase, de la clase *Ventana*.

La primera sentencia del constructor de la clase derivada es una llamada al constructor de la clase base mediante la palabra reservada **super**. La llamada

```
super(x, y, w, h);
```

inicializa los cuatro miembros dato de la clase base *Ventana*: *x*, *y*, *ancho*, *alto*. A continuación, se inicializa los miembros dato de la clase derivada, y se realizan las tareas de inicialización que sean necesarias. Si no se llama explícitamente al constructor de la clase base Java lo realiza por nosotros, llamando al constructor por defecto si existe.

La función miembro denominada *desplazar* cambia la posición de la ventana, añadiéndoles el desplazamiento.

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Redefine la función miembro *mostrar* para mostrar una ventana con un título.

```
public void mostrar(){
    super.mostrar();
    System.out.println("titulo     : "+titulo);
}
```

En la clase derivada se define una función que tiene el mismo nombre y los mismos parámetros que la de la clase base. Se dice que redefinimos la función *mostrar* en la clase derivada. La función miembro *mostrar* de la clase derivada *VentanaTitulo* hace una llamada a la función *mostrar* de la clase base *Ventana*, mediante

```
super.mostrar();
```

De este modo aprovechamos el código ya escrito, y le añadimos el código que describe la nueva funcionalidad de la ventana por ejemplo, que muestre el título.

Si nos olvidamos de poner la palabra reservada **super** llamando a la función *mostrar*, tendríamos una función recursiva. La función *mostrar* llamaría a *mostrar* indefinidamente.

```
public void mostrar(){ //¡ojo!, función recursiva
    System.out.println("titulo     : "+titulo);
    mostrar();
}
```

5.3 Definición de una clase derivada.

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.

Cabecera/Declaración de clase

{

Cuerpo de clase

}

En la clase FactorialSinBucle estudiada en módulos anteriores podemos identificar la

declaración y cuerpo de la clase:

public class FactorialSinBucle

{

```
 public static void main(String[] args)
 {
 final int entero = 5;
 int factorial;
 factorial = 5*4*3*2*1;
 System.out.println(entero + "! = " + factorial);
 }
}
```

5.4 Clases abstractas.

Se fija un conjunto de métodos y atributos que permitan modelar un cierto concepto, que será refinado mediante la herencia.

Métodos abstractos:

sólo cuentan con la declaración y no poseen cuerpo de definición

la implementación es específica de cada subclase

Toda clase que contenga algún método abstracto (heredado o no) es abstracta. Puede tener también métodos efectivos.

Tiene que derivarse obligatoriamente

NO se puede hacer un new de una clase abstracta. SI deben definir los constructores.

Todas las clases heredan directa o indirectamente de la clase Object, raíz de la jerarquía.

Toda clase tiene disponibles sus métodos:

public final Class getClass() → clase que representa el tipo del objeto

public boolean equals(Object obj) → igualdad de valores

public String toString() → Devuelve la representación del obj en un String

protected Object clone() → devuelve una copia del objeto

public int hashCode() → devuelve un código que identifica de manera única al objeto

protected void finalize() → relacionado con liberar memoria

Hay que redefinir equals, toString, hashCode y clone para adaptarlos**.**

5.5 Definición de herencia múltiple.

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de más de una superclase. Esto contrasta con la **herencia simple**, donde una clase sólo puede heredar de una superclase.

Lenguajes que soportan herencia múltiple en su mayor parte son: C++, Centura SQL Windows, CLOS, Eiffel, Object REXX, Perl y Python.

La herencia múltiple permite a una clase tomar funcionalidades de otras clases, como permitir a una clase llamada MusicoEstudiante heredar de una clase llamada Persona, una clase llamada Músico, y una clase llamada Trabajador. Esto puede ser abreviado como MusicoEstudiante : Persona, Músico, Trabajador.

5.6 Implementación de herencia múltiple.

Un hecho natural es que una persona tenga más de un pariente mayor, esta situación también se puede dar en la herencia de clases, naturalmente este tipo de herencia involucra un mayor grado de complejidad en el lenguaje, sin embargo, el beneficio logrado es substancial. Considere por ejemplo, la clase llamada MesaRedonda la cual tiene las propiedades de una Mesa y también tiene las propiedades de un Circulo, de acuedo a la siguiente estructura.

Mesa

altura : real = initval

Mesa()

alto()

Circulo

radio : real = initval

Circulo()

area()

MesaRedonda

color : entero = initval

MesaRedonda()

Color()

Veamos ahora la estructura de clases en C++.

```
// Programa PLCP84.CPP
// Herencia Múltiple.
#include <math.h>
#include <stdio.h>

class Circulo {
 float radio;
public:
 Circulo(float r) { radio = r; }
```

```
 float area() { return M_PI*radio*radio; }
};


class Mesa {
 float altura;
public:
 Mesa(float a) { altura = a; }
 float alto() { return altura; }
};
```

5.7 Reutilización de la definición de paquetes / librerías.

El Java API (Application Program Interfase) es un conjunto de librerías que permiten el desarrollo de aplicaciones en Java, brinda funciones de uso común para el programador como por ejemplo:

Creación y manejo de elementos de GUI

Manejo de archivos

Funciones de red

Comunicación entre programas

5.8 Clases genéricas (Plantillas).

Las clases-plantilla, clases genéricas, o generadores de clases, son un artificio C++ que permite definir una clase mediante uno o varios parámetros. Este mecanismo es capaz de generar la definición de clases (instancias o especializaciones de la plantilla) distintas, pero compartiendo un diseño común. Podemos imaginar que una clase genérica es un constructor de clases, que como tal, acepta determinados argumentos (no confundir con el constructor de-una-clase, que genera objetos).

```
class mVector {                  // definición de la clase mVector
  int dimension;
  public:
  Vector* mVptr;
  mVector(int n = 1) {           // constructor por defecto
    dimension = n;
    mVptr = new Vector[dimension];
  }
  ~mVector() { delete [] mVptr; } // destructor
  Vector& operator[](int i) {     // operador subíndice
    return mVptr[i];
  }
  void showmem (int);             // función auxiliar
};

void mVector::showmem (int i) {
  if((i >= 0) && (i <= dimension)) mVptr[i].showV();
  else cout << "Argumento incorrecto! pruebe otra vez" << endl;
}
```

El sistema de plantillas permite definir una clase genérica que instancie versiones de mVector para matrices de cualquier tipo especificado por un parámetro. La ventaja de este diseño parametrizado, es que cualquiera que sea el tipo de objetos utilizados por las especializaciones de la plantilla, las operaciones básicas son siempre las mismas (inserción, borrado, selección de un elemento, etc).

Definición

La definición de una clase genérica tiene el siguiente aspecto:

```
template<lista-de-parametros> class nombreClase {  // Definición
   ...
};
```

Una clase genérica puede tener una declaración adelantada para ser declarada después:

```
template<lista-de-parametros> class nombreClase;   // Declaración
...
template<lista-de-parametros> class nombreClase {  // Definición
   ...
};
```

pero recuerde que debe ser definida antes de su utilización [5], y la regla de una sola definición.

Observe que la definición de una plantilla comienza siempre con **template <...>**, y que los parámetros de la lista**<...>** no son valores, sino tipos de datos. En la página adjunta se muestra la gramática C++ para el especificador **template**

La definición de la clase genérica correspondiente al caso anterior es la siguiente:

```
template<class T> class mVector {     // L1 Declaración de plantilla
  int dimension;
  public:
  T* mVptr;
  mVector(int n = 1) {               // constructor por defecto
    dimension = n;
    mVptr = new T[dimension];
  }
  ~mVector() { delete [] mVptr;  }    // destructor
  T& operator[](int i) { return mVptr[i]; }
  void showmem (int);
};

template<class T> void mVector<T>::showmem (int i) {    // L16:
  if((i >= 0) && (i <= dimension)) mVptr[i].showV();
  else cout << "Argumento incorrecto! pruebe otra vez" << endl;
}
```

Observe que aparte del cambio de la declaración en L1, se han sustituido las ocurrencias de Vector (un tipo concreto) por el parámetro T. Observe también la definición de showmem() en L16, que se realiza off-line con la sintaxis de una función genérica .

El especificador **class** puede ser sustituido por **typename**, de forma que la expresión L1 puede ser sustituida por:

```
tamplate<typename T> class mVector {      // L1-bis
  ...
};
```