Programación II

Carla TARAMASCO

Profesora e Investigadora

DECOM & CNRS

mail : alumnos_uv@yahoo.cl

16 avril 2013

Herencia

- Herencia
  - Acceso a una clase derivada
  - Construcción de objetos derivados
  - Sobreescritura y sobrecarga de metodos
  - Polimorfismo
  - Super clase object
  - Miembros protected
  - Clases y metodos de finalización
  - Clases abstractas
  - Clases Interfaces

Herencia

Definiciones

La herencia es uno de los conceptos fundamentales de la POO, esta al origen de la reutilización.
La herencia permite definir una nueva clase llamada **derivada** a partir de una clase existente llamada clase **base**.
Esta nueva clase hereda automáticamente las funcionalidades de la clase base (métodos y atributos), los que podrá modificar o completar libremente.
Es posible crear muchas clases derivadas a partir de una clase base.

Herencia

Nociones iniciales

Tenemos una clase base Punto :

```
class Punto{
        private int x,y;
        public void inicializar (int abs, int ord){
                x=abs;
                y=ord;}
        public void desplazar(int dx, int dy){
                x+=dx;
                y+=dy;}
        public void mostrar(){
                System.out.println("punto de coordenadas : "+x+" " + y);}
}
```

Definir una clase derivada de la clase Punto que se llame : **PuntoColor**. Para esto, en java usaremos la palabra clave **extends**. Esta clase debe manipular puntos coloreados en un plano.
Dentro de la clase **PuntoColor** cree un método llamado **color** encargado de definir el color del punto.
Cree un metodo que muestre el color del punto.

Herencia

Nociones iniciales

Tenemos una clase derivada de la Punto llamada PuntoColor :

```
class PuntoColor extends Punto{
        private int color;
        public void colorM (int color){
                this.color=color;}
        public void mostrarC(){
                this.mostrar();
                System.out.println("El color del punto es :"+color);}}

public class PuntoHerencia{
        public static void main(String args []){
                PuntoColor pc=new PuntoColor();
                pc.mostrar();
                pc.inicializar(3,5);
                pc.colorM(3);
                pc.mostrarC();
                pc.desplazar(2,3);
                pc.mostrar();
                PuntoT p1=new PuntoT();
                p1.inicializar(4,7);
                p1.mostrar();}}
```

Al usar extends Punto se le especifica al compilador que la clase PuntoColor es una clase derivada de la clase Punto.

Herencia

Nociones iniciales

Un objeto del tipo PuntoColor puede acceder a :

- A los métodos públicos de PuntoColor
- A los métodos públicos de Punto

Acceso de una clase derivada

a la clase base

Si bien una clase derivada hereda los miembros (atributos y métodos) de una clase base, el acceso que tenga a estos dependerá de si estos son privados o públicos.

- **Una clase derivada no accede a los miembros privados** : Un método de la clase derivada no puede acceder a miembros privados de su clase base.
- **Una clase derivada accede a los miembros públicos** : tal como si hubiesen sido declarados en la clase la misma clase derivada. (Ejemplo precedente).

Acceso de una clase derivada

a la clase base

- Las clases de base pueden llamarse también superclases. De la misma manera que las clases derivadas puedes llamarse subclases.
- Si trabajamos con múltiples clases (cada una en archivos fuentes separados), se debe compilar primero la superclase y luego las subclases.
- El acceso a las clases no publicas será restringido si no pertenecen al mismo paquete. Una clase sin la palabra clave *public* será accesible solo por las clases del mismo paquete.

Acceso de una clase derivada

a los miembros de la clase base

En resumen :

- Los miembros públicos de la superclase quedan como miembros públicos para la subclase. Por esa razón, en el ejemplo precedente, pudimos aplicar el método inicializar a un objeto de tipo PuntoColor.

Modifique el método mostrar de la clase PuntoColor para que ademas muestre el valor de x e y usando :
System.out.println("punto de coordenadas : "+x+" " + y) ;.
Que ocurre ?

Acceso de una clase derivada

a los miembros de la clase base

```
public void mostrarC(){
mostrar();
System.out.println("El color del punto es :"+color);}}
```

```
public void mostrarC(){
this.mostrar();
System.out.println("El color del punto es :"+color);}}
```

Si usamos *this.mostrar()*, lo que hace es aplicar el metodo mostrar al objeto (de tipo PuntoColor) que llamo al metodo *mostrarC*.
Cree un nuevo metodo inicializar que se atribuya las coordenadas y el color a un punto.

```
public void inicalizarC(int x,int y, int color){
inicializar(x,y);
this.color=color}}
```

Construcción e inicialización

de objetos derivados

Llamar a constructores : La creación de un objeto usando *new* llama al constructor (que tiene el mismo nombre de la clase). Si la clase no dispone de un constructor usara un pseudo-construtor por defecto.

```
class Punto{
        public Punto(int x, int y){....}
        private int x,y;}

class PuntoColor extends Punto{
        public PuntoColor(int x, int y, int color){....}
        private int color;}
```

Si necesita inicializar ciertos campos de la superclase sera necesario disponer de las funciones de alteración (set) o recurrir al constructor de la superclase.

Construcción e inicialización

de objetos derivados

El constructor de PuntoColor podrá :

- Inicializar el campo color, accesible dado que es miembro de PuntoColor ;
- Llamar al constructor de Punto para inicializar los campos x e y.

Regla : Si un constructor de una subclase llama a un constructor de una superclase, debe ser la primera instrucción del constructor y debe usar la palabra clave **super**. Solo es posible llamar al constructor de la superclase inmediatamente superior usando **super**.

Construcción e inicialización

de objetos derivados

```
public PuntoColor(int x,int y, int color){
        super(x,y);
        this.color=color;}
```

Reemplace los metodos inicializar de la clase Punto y PuntoColor por constructores.

Para inicializar la parte heredada de una superclase que no tiene constructor se podrá llamar al constructor por defecto usando *super();*

Construcción e inicialización

de objetos derivados

```
public PuntoColor(int x,int y, int color){
        super(x,y);
        this.color=color;}
```

Reemplace los metodos inicializar de la clase Punto y PuntoColor por constructores.

Para inicializar la parte heredada de una superclase que no tiene constructor se podrá llamar al constructor por defecto usando *super ();*

Construcción e inicialización

Que pasa si una de las clases no tiene constructor ?

- La clase base no tiene constructor : en estos casos se llama al constructor por defecto de la clase base usando*super () ;*. Asi la parte heredada de la superclase sera inicializada.

```
class A {} // clase a sin constructor
class B extends A {
        public B(){ //constructor de B
                super()}} ?// llama al constructor por defecto de A
```

Construcción e inicialización

Que pasa si una de las clases no tiene constructor ?

- La clase de derivada no tiene constructor : en estos casos la clase base debe :
  1. tener un constructor publico sin argumentos o
  2. no tener constructor

```
class A { // clase a sin constructor
public A(){} //constructor de A
        public A(int n){}} //constructor de A
class B extends A {
         //sin constructor
        B b=new B //llama al constructor sin argumentos de A

class A { // clase a sin constructor
        public A(int n){}} //constructor de A
class B extends A {
        //sin constructor
        B b=new B //error!

class A { // clase a sin constructor
        //sin constructor
        class B extends A {
        //sin constructor
        B b=new B //llama al constructor por defecto de A
```

Ejercicio de constructores y herencia

```
class Punto{
        public Punto(int x, int y){
                this.x=x;
                this.y=y;}
        public void desplazar(int dx, int dy){
                x+=dx;
                y+=dy;}
        public void mostrar(){
                System.out.println("punto de coordenadas : "+x+" " + y);}
        private int x,y;}

class PuntoColor extends Punto{
        public PuntoColor(int x, int y, int color){
                super(x,y);
                this.color=color;}
        public void mostrarC(){
                this.mostrar();
                System.out.println("El color del punto es :"+color);}
        private int color;}

public class PuntoHerenciaConstr{
        public static void main(String args []){
                PuntoColor pc=new PuntoColor(3,5,7);
                pc.mostrar();
                pc.mostrarC();
                PuntoColor pc2=new PuntoColor(5,7,8);
                pc2.mostrar();
                pc2.desplazar(2,3);
                pc2.mostrar();}}
```

Ejercicio : Constructores y Herencia

Cree una superclase Fruta con : tipoFruta (string) y precio (float). Cree el constructor de dicha clase y un método mostrar que muestre los 2 atributos.
Cree una subclase Manzana con : rebanada (int), tipoFruta (string) y precio (float). Cree el constructor de dicha clase y un metodo mostrar que despliegue los valores.

Ejercicio : Constructores, Herencia y Encapsulamiento

```
class Fruta{
    float precio;
    String tipoFruta;
    public Fruta(float p, String v){
      precio=p;
      tipoFruta=v;}

     public void mostrar()
    {System.out.println("Variedad: "+tipoFruta+", Precio: "+precio);}}

class Manzana extends Fruta
{       int rebanada;
    public Manzana(float p, String v,int rebanada){
                super(p,v);
                this.rebanada=0;}

        public void mostrar()
    {System.out.println("Variedad: "+tipoFruta+", Precio: "+precio+", Rebanada:"+rebanada);}}

public class HerenciaFruta {
   public static void main(String[] args){
       Fruta platano=new Fruta(1000,"El Platano");
       Manzana Mroja=new Manzana(500,"Manzanas rojas", 2);
       Mroja.rebanada=1;
       Mroja.mostrar();
       platano.mostrar();}}
```

Ejercicio : Constructores y Herencia

```
class Fruta{
    private float precio;
    private String tipoFruta;
    public Fruta(float p, String v){
      precio=p;
      tipoFruta=v;}

    public void mostrar()
    {System.out.println("Variedad: "+tipoFruta+", Precio: "+precio);}}

class Manzana extends Fruta
{private int rebanada;
    public Manzana(float p, String v,int rebanada)
    {super(p,v);
        this.rebanada=rebanada;}
        void mostrarM(){
        mostrar();
    System.out.println("Rebanada:"+rebanada);}
        public void setRebanada(int valor) {
        this.rebanada=valor;}}
public class HerenciaFrutaPrivate{
   public static void main(String[] args){
       Fruta platano=new Fruta(1000,"El Platano");
       Manzana Mroja=new Manzana(500,"Manzanas rojas", 2);
       Mroja.mostrarM();
           Mroja.setRebanada(8);
       Mroja.mostrarM();
       platano.mostrar();}}
```

Ejercicio : Constructores, Herencia y Encapsulamiento

Cree una superclase Persona con : rut (string) y nombre (string).
Cree el constructor de dicha clase .
Cree una subclase Estudiante con : rol (string), rut (string) y nombre (string). Cree el constructor de dicha clase.
Cree un metodo principal que llame a las clases y muestre en pantalla sus atributos.

Ejercicio de constructores y herencia

```
public class Persona {
        private String rut;
        private String nombre;

        public Persona(String rut, String nombre) {
                this.rut = rut;
                this.nombre = nombre;
        }

        public String getRut() {
                return rut;
        }

        public String getNombre() {
                return nombre;
        }}
```

Ejercicio de constructores y herencia

```
public class Estudiante extends Persona {
        private String rol;

        public Estudiante(String rol, String rut, String nombre) {
                super(rut,nombre);
                this.rol = rol;}

        public String getRol() {
                return rol;}}
public class EjemploHerencia {
        public static void main(String[] args) {
                Persona profesora = new Persona("8.976.400-4", "Carla");
                Estudiante estudiante = new Estudiante("2234512-1","10.336.400-4","Jorge");
                System.out.println("Nombre de persona : "+ profesora.getNombre()+
                                                        " Rut "+profesora.getRut());
                System.out.println("Nombre de estudiante "+estudiante.getNombre()+
                                                        " Rol : "+estudiante.getRol());}}
```

Inicalización de un objeto derivado

Recordatorio : Creación de un objeto

- asignación de memoria.
- inicialización por defecto de los campos.
- inicialización explicita de los campos.

Inicialización de un objeto derivado

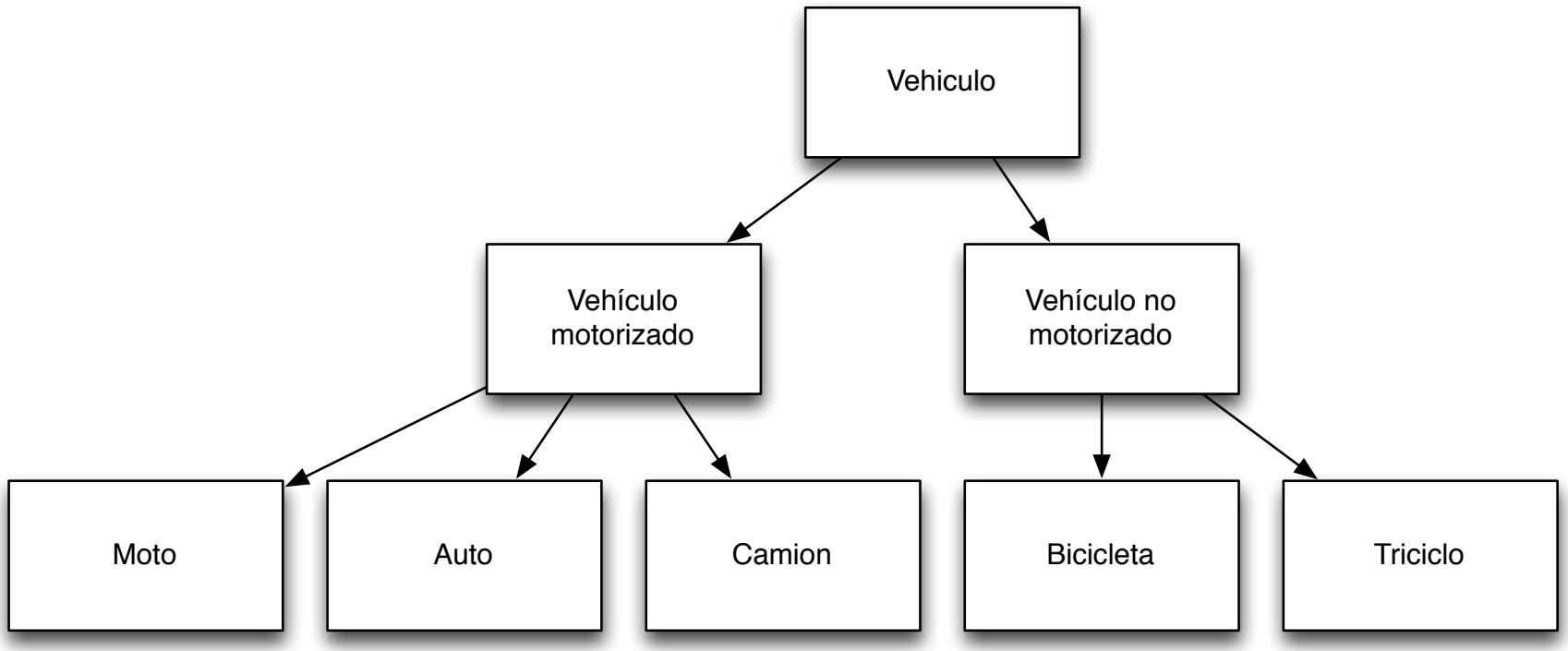
Creación de un objeto derivado

- asignación de memoria, considerando los campos heredados y los propios.
- inicialización por defecto de los campos, considerando los campos heredados y los propios
- inicialización explicita de los campos, considerando los campos heredados y los propios por lo que se ejecuta los bloques de inicialización de la clase base.
- inicialización explicita de los campos de la clase derivada.

Derivaciones sucesivas

- De una clase base pueden derivar varias clases diferentes.
- La noción de clase de base y clase derivada es relativa dado que la clase derivada puede a la vez servir de clase base para otras clases.

Herencia


Vehiculo
Vehículo motorizado
Vehículo no motorizado
Moto
Auto
Camion
Bicicleta
Triciclo

Polimorfismo

El polimorfismo sugiere múltiples formas. El polimorfismo es un concepto muy importante dentro de POO. En java es la habilidad de una variable por referencia de cambiar su comportamiento en función de que instancia de objeto posee. Esto permite tratar de la misma manera, como objetos de la súperclase, a múltiples objetos de la subclase, seleccionando en cada caso los métodos apropiados. El polimorfismo se puede establecer mediante la sobrecarga, sobre-escritura y la ligadura dinámica.

Sobreescritura y sobrecarga de miembros

- Una clase derivada podrá sobrecargar un método de la clase base. Estos nuevos métodos serán usados solo por la clase derivada y sus descendientes.
- Una clase derivada podrá sobreescribir (redefinir) un método de la clase base. En este caso, no solo serán métodos con el mismo nombre como en la sobrecarga sino también con los mismo argumentos de entrada (cantidad y tipo) y el mismo tipo de valor de retorno.

La sobrecarga permite acumular métodos con el mismo nombre la Sobreescritura permite sustituir un método con otro.

Sobreescritura de métodos

Un objeto de una clase derivada puede acceder a todos los miembros públicos de la clase de base.

```
class Punto{
        private int x,y;
        public void mostrar(){
                System.out.println("punto de coordenadas : "+x+" " + y);}}

class PuntoColor extends Punto{
        private int color;}

public class Main{
        public static void main(String args []){
                Punto p, PuntoColor pc;
```

Si llamamos p.mostrar() o pc.mostrar(). Obtendremos las coordenadas del punto *p* o del punto *pc* respectivamente. Pero no obtendremos el color del punto color. Por esto, hemos creado anteriormente dos métodos con distinto nombre.

Sobreescritura de métodos

En java es posible sobreescribir en la clase derivada un método de la clase base con los mismos argumentos de entrada y el mismo tipo de retorno. Dentro de la clase derivada se llamará al nuevo método *sobrescrito* que esta dentro de ella ocultando de alguna manera el método de la clase base.

```
class PuntoColor extends Punto{
        private int color;
        public void mostrar(){
                mostrar();
                System.out.println("el color del punto es : "+color);}}
```

Sobreescritura de métodos

El código anterior provoca una llamada recursiva al método mostrar de la clase PuntoColor y no llama al método mostrar de la clase Punto. Para eso es necesario especificar que se quiere llamar al método mostrar de la clase base.

```
class PuntoColor extends Punto{
        private int color;
        public void mostrar(){
                super.mostrar();
                System.out.println("el color del punto es : "+color);}}
```

Sobrecarga y Sobreescritura de métodos

Puede usarse sobrecarga de métodos de la clase base en la clase derivada.

```
class A{
        ....
        public void f (int n) {...}
        public void f (float x) {...}
}
class B extends A{
        ....
        public void f (int n) {...} //redefinicion de f(int n)
        public void f (double y) {...} // sobrecarga de f
}
.....
A a; B b;
int n;float x; double y;
...
a.f(n); //sobrecarga
a.f(x); //sobrecarga
a.f(y); //error
b.f(n); //redefinicion
b.f(x); //sobrecarga
b.f(y); //sobrecarga de A
```

Restricciones en la Sobreescritura

- debe tener los mismo argumentos (tipo y cantidad) de entrada y el mismo tipo de retorno

```
class A{
        public int f (int n) {...}}
class B extends A{
        public float f (int n) {...}} //redefinicion de f(int n)
```

- no debe disminuir los derecho de acceso del método

```
class A{
        public int f (int n) {...}}
class B extends A{
        private float f (int n) {...}} //redefinicion de f(int n)
```

- puede aumentar los permisos de acceso.

```
class A{
        private int f (int n) {...}}
class B extends A{
        public float f (int n) {...}} //redefinicion de f(int n)
```

Restricciones en la Sobreescritura

Un método de clase (static) no puede ser sobrescrito en una clase derivada dado que el tipo de objeto que llama al método permite elegir entre el método de la clase base y el método de la clase derivada.
Recuerde que un método de clase puede ser llamado sin un objeto, por lo cual, la elección del método no es posible.

Duplicación de campos

Una clase derivada puede definir un campo con el mismo nombre que un campo de la clase base.

```
class A{
        public int n=8;
        public void mostrar(){
                System.out.println ("En A n= "+ n);}}
class B extends A{
        public float n=5.6f;
    public void mostrar(){
                n=5.34f;
                super.n=4;
                System.out.println ("En B n= "+ n);
            super.mostrar();}}
public class Main{
        public static void main(String args []){
                A a=new A();
                B b=new B();
                System.out.println ("a.n= "+ a.n);
                System.out.println ("b.n= "+ b.n);
            b.mostrar();}}
```

Polimorfismo

Permite manipular los objetos sin conocer su tipo.
Por ejemplo, se puede crear un arreglo de objetos unos de tipo Punto y otros PuntoColor y llamar al método mostrar para cada objeto del arreglo. Cada objeto actuará en función de su tipo. Esto es inducido por la herencia.
Un objeto PuntoColor es también un Punto por lo cual puede ser tratado como un Punto (esta relación no es reciproca).

Polimorfismo

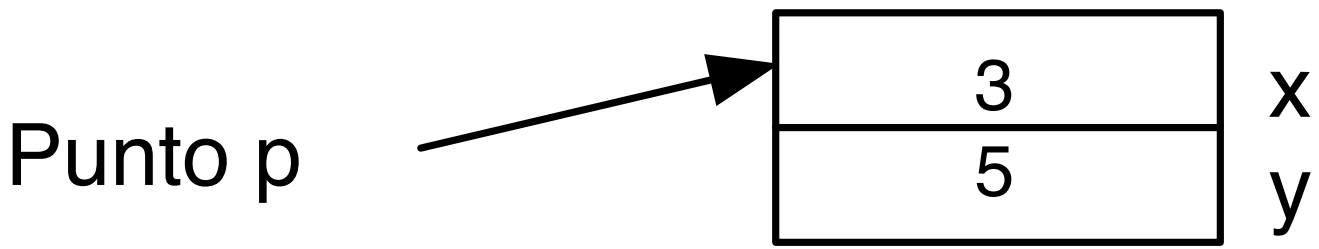
```
class Punto{
        public Punto (int x,int y){
                this.x=x;this.y=y;}
        private int x,y;
        public void desplazar(int dx, int dy){
                x+=dx;
                y+=dy;}
        public void mostrar(){
                System.out.println("punto de coordenadas : "+x+" " + y);}}

class PuntoColor extends Punto{
        public PuntoColor (int x, int y, int color){
                super(x,y);
                this.color=color; }
        private int color;
        public void mostrar(){
                super.mostrar();
                System.out.println("El color del punto es :"+color);}}

public class MainPoli{
        public static void main(String args []){
                Punto p=new Punto(3,5);
                p.mostrar();
                PuntoColor pc=new PuntoColor(4,3,7);
                //p=pc; //p del tipo Punto referencia a un objeto tipo PuntoColor
                pc=(PuntoColor)p;
                p.mostrar();
                p=new Punto(9,8);
                p.mostrar();}}
```

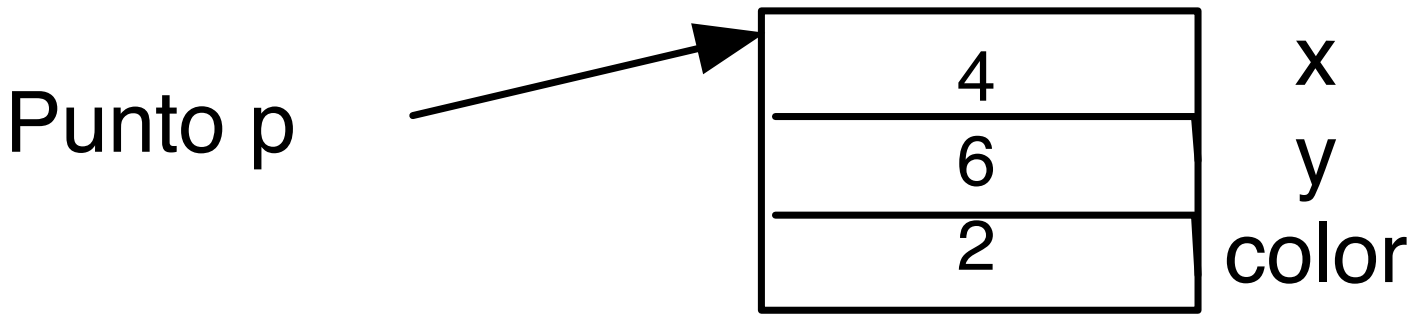
Polimorfismo

Punto p ;



p=new PuntoColor(4,6,2) ;

Polimorfismo



Polimorfismo

Java permite asignar a una variable objeto no solo la referencia del tipo correspondiente sino también la referencia a un objeto de un tipo derivado. Encontramos así, una compatibilidad por asignación entre un tipo clase derivada y un tipo de clase base.
Si ejecutamos :

```
Punto p = new Punto(3,5) ;
p.mostrar() ;
p=new PuntoColor(4,5,2) ;
p.mostrar() ;
```

La variable p es del tipo Punto mientras que el objeto referenciado por p es del tipo PuntoColor. La variable p llamara a mostrar de la clase PuntoColor.

Resumen : Polimorfismo

- compatibilidad : existe una conversion implicita de una referencia a un objeto de una clase derivada a una referencia de un objeto de una clase base (ascendente).
- ligado dinámico de métodos

El polimorfismo permite obtener un comportamiento adecuado de cada tipo de objeto, sin la necesidad de conocer su tipo.

Ejercicio : Polimorfismo

Haga un arreglo que contenga 2 objetos del tipo Punto y dos objetos del tipo PuntoColor. Muestre los valores en pantalla.

Ejercicio : Polimorfismo

```
class Punto{
        public Punto (int x,int y){
                this.x=x;this.y=y;}
        private int x,y;
        public void mostrar(){
                System.out.println("punto de coordenadas : "+x+" " + y);}}

class PuntoColor extends Punto{
        public PuntoColor (int x, int y, int color){
                super(x,y);
                this.color=color; }

        private int color;
        public void mostrar(){
                super.mostrar();
                System.out.println("El color del punto es :"+color);}}

public class MainPolimorf{
        public static void main(String args []){
                Punto [] tabPuntos=new Punto[4];
                tabPuntos[0]=new Punto(0,2);
                tabPuntos[1]=new PuntoColor(1,4,5);
                tabPuntos[2]=new PuntoColor(2,6,8);
                tabPuntos[3]=new Punto(7,8);
                for (int i=0;i<tabPuntos.length;i++)
                        tabPuntos[i].mostrar();
        }}
```

Conversion explicita

Los objetos derivados son compatibles con los objetos de clases ascendentes, pero no ocurre lo mismo inversamente.

```
class Punto {....}
class PuntoColor {....}
....
PuntoColor pc;
pc=new Punto(....) // error!!

Punto P;
PuntoColor pc1=new PuntoColor();
pc1=p;  // error!!

pc1=(PuntoColor) p;}
```

La super clase Object

De la clase Object derivan todas las clases simples. Por ejemplo, la clase Punto : **class Punto** es lo mismo que decir : **class Punto extends Object** . Una variable de tipo Object puede ser usada para referenciar un objeto de cualquier tipo :

```
Punto p=new Punto();
PuntoColor pc =new PuntoColor();
Flor f = new Flor();
Object o;
.....
o=p;
o=pc;
o=f;
```

Esto es conveniente si necesitamos trasmitir a un método una referencia sin conocer su tipo. Evidentemente, cualquier se quiera aplicar un método particular de un objeto referenciado por una variable de tipo Object será necesario hacer una conversión explícita.

La super clase Object

Si la clase Punto tiene un método desplazar.

```
Punto p=new Punto();
Object o;
.....
o=p;
o.desplazar(); //error!!
Punto p1 = (Punto)o;
p1.desplazar();
```

Si bien, el objeto referenciado por *o* es de un tipo que contiene un método *f*, es necesario que dicho método exista en la clase Object.

Metodos de la clase Object

toString

Este método de la clase Object entrega una cadena de caracteres con :

- el nombre de la clase
- la dirección del objeto en hexadecimal precedido de @

```
class Punto{
        public Punto(int abs, int ord){
                x=abs;y=ord;}
        private int x,y;}
public class ClaseToString{
        public static void main (String arg[]){
                Punto a = new Punto(3,5);
                Punto b = new Punto(4,6);
                System.out.println("a = " + a.toString());
                System.out.println("b = " + b.toString());
                System.out.println("llamada automatica = " + b);}}
```

Metodos de la clase Object

toString

- El nombre de la clase : nombre de la clase del objeto que llama a toString.
- Puede ser llamada automaticamente en el caso de necesitar una conversion implicita a cadena de caracteres.

Metodos de la clase Object

equals

Este metodo compara las direcciones de dos objetos.
Object o1= new Punto(1,3) ;
Object o1= new Punto(1,3) ;
o1.equals(o2) ;
Será false.

Miembros protegidos

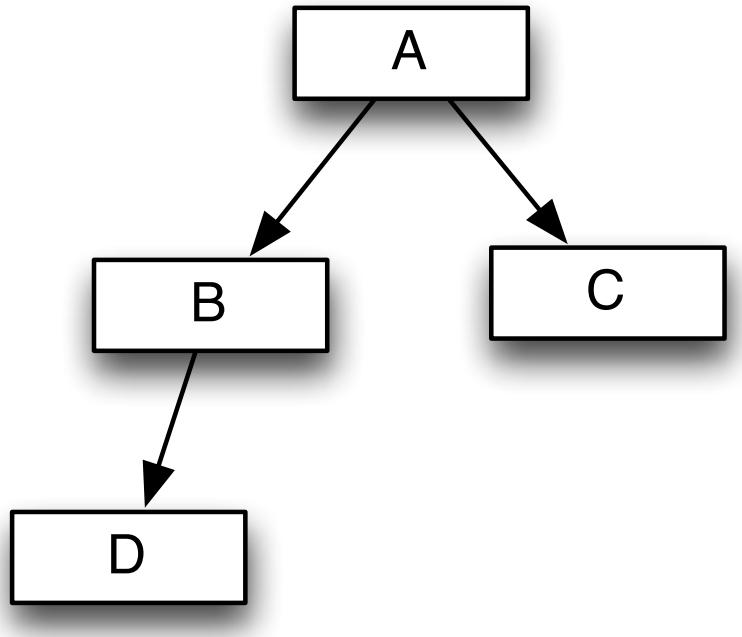
protected

Tipo de acceso protected actua en :

- empaquetamiento de clases
- clases derivadas

Un miembro protegido es accesible a las clases del mismo paquete así que a las clases derivadas (que pueden pertenecer o no al mismo paquete).
Considere :
class A {
protected int n ;}

Herencia



- *B* accede a *n* de *A*
- *D* accede a *n* de *B* o de *A*
- *C* no accede a *n* de *B* (excepto si *B* y *C* están en el mismo paquete)

Clases y metodos finales

final

Si se aplica la palabra clave final a variables locales o campos de una clase, esta prohibe la modificación de su valor. Esta palabra puede también aplicarse a una clase o método pero con una significación totalmente diferente.

- Un método declarado final no puede ser sobreescrito en una clase derivada.
- Una clase declarada final no puede puede ser derivada.

Interfaces

Un interface es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes. Una interface define las cabeceras de un cierto numero de métodos.

- Una clase podrá implementar diversas interfaces
- la noción de interface se superpone a la derivación
- las interfaces puedes derivarse
- es posible usar variables de tipo interface

Definir una interface

interfaces

Una interface puede tener los mismo permisos que una clase. Por esencia las interfaces son publicas y sus métodos son abstractos.

```
public interface I{
        void f(int n);
        void g;}
```

Implementar una interface

interfaces

Al definir una clase se puede precisar que implementa una interface dada usando la palabra clave *implements*. Una clase puede implementar diferentes interfaces.

```
class A implements I,I1{....}
public interface I{
        void f(int n);}
public interface I1{
        int h();}
```

Realizar una programa que tenga una interface Afichar con un metodo mostrar. Dos clases Entero y Flotante implementando esta interface. Crear un arreglo heterogéneo de referencias de tipo Afichar que se llenaran con instancias de objetos Flotante y Entero.

Ejercicio

```
interface Afichar{
        void mostrar();}
class Entero implements Afichar{
        public Entero(int n){ valor=n;}
        public void mostrar(){
                System.out.println("El valor es : : " + valor);}
        private int valor;      }
class Flotante implements Afichar{
        public Flotante(float x){ valor=x;}
        public void mostrar(){
                System.out.println("El valor es : : " + valor);}
        private float valor;    }
public class Tabla2 {
        public static void main (String arg[]){
                Afichar[] tab;
                tab=new Afichar[3];
                tab[0]=new Entero(24);
                tab[1]=new Flotante(3.4f);
                tab[2]=new Entero(24);
                for (int i=0;i<3;i++)
                        tab[i].mostrar();}}
```

Clases abstractas

Clases abstractas

Una clase abstracta es una clase que no permite instanciar objetos. Una clase abstracta permite definir en una clase base las funcionalidades comunes a todas las clases derivadas, si consideramos que una clase abstracta no implementa ningún método y ningún campo eso será una interface.
Una clase abstracta puede servir solo como clase base, se usa como interface de las clases que harán conversión hacia arriba (Podemos describir el upcasting como la acción de declarar una variable de una clase base (abstracta en la mayoría de los casos), pero instanciando una implementación de la misma (comportamiento polifórmico).

Clases abstractas

Clases abstractas

En una clase abstracta se pueden encontrar métodos y campos que heredan todas las clases derivadas.
abstract class A
Será posible declarar : *A a*;, pero no sera posible $a = new\ A()$;
Sin embargo se puede derivar A e instanciar un objeto de la clase derivada.
class B extends A
$A\ a = new\ B()$;

Métodos abstractos

Métodos abstract

No se realiza la definición del método sino que solo que entrega la lista de argumentos de entrada y el tipo de valor de salida.
Ejemplo : *public abstract void g* (*int n*);

Reglas de abstract

abstract

1. Cuando una clase tiene uno o varios métodos abstractos, la clase es abstracta aunque no sea indicado explícitamente en su declaración.
2. Un método abstracto debe obligatoriamente ser declarado publico, dado que se espera que sea sobrescrito en una clase derivada
3. En la cabecera de la declaración de un método abstracto los nombres de los argumentos deben figurar
4. Una clase derivada de una clase abstracta no esta obligada a sobreescribir todos los métodos abstractos (puede no sobreescribirlos)
5. Una clase derivada de una clase abstracta puede ser definida abstracta

Interes de abstract

abstract

Las clases abstractas facilitan la concepción OO dado que en una clase abstracta se pueden poner todas las funcionalidades que se desea disponer para todas las clases descendientes. Esto puede ser a través de :

- implementando completamente los métodos y campos cuando son comunes a todos los descendientes
- como interface de métodos abstractos

Ejercicio

Declare una clase abstracta Afichar con un metodo abstracto mostrar. Dos clases Entero y Flotante derivan de esta clase. El metodo main usa un arreglo heterogéneo de objetos de tipo Afichar que completa instanciando objetos de tipo Entero y Flotante.

Ejercicio

```
abstract class Afichar{
        abstract public void mostrar();}
class Entero extends Afichar{
        public Entero(int n){ valor=n;}
        public void mostrar(){
                System.out.println("El valor es : : " + valor);}
        private int valor;      }
class Flotante extends Afichar{
        public Flotante(float x){ valor=x;}
        public void mostrar(){
                System.out.println("El valor es : : " + valor);}
        private float valor;    }
public class Tabla {
        public static void main (String arg[]){
                Afichar[] tab;
                tab=new Afichar[3];
                tab[0]=new Entero(24);
                tab[1]=new Flotante(3.4f);
                tab[2]=new Entero(24);
                for (int i=0;i<3;i++)
                        tab[i].mostrar();}}
```

Clase abstract e interfaces

Resumen

- Clase Abstracta :
  - Contiene tanto métodos ejecutables como métodos abstractos
  - Una clase puede extender solo una clase
  - Puede tener variables de instancia, constructores y cualquier tipo de visibilidad (public, private, protected)
- Interface :
  - No contiene código de implementación
  - Una clase puede implementar *n* numero de interfaces
  - No puede tener variables de instancia, constructores y solo puede tener métodos publicos o package