Programación
Tema 7: Relaciones entre clases

{PROG} dit UPM

Contenidos

- **Polimorfismo**
- Implementación de interfaces
- Uso de interfaces
- Jerarquías de interfaces
- Extensión
- Jerarquía de extensión

{PROG} dit UPM

Polimorfismo

- **Polimorfismo**: elementos parecidos pero distintos se usan de igual forma
  - **Clases que tienen métodos públicos con la misma cabecera**
  - Sus objetos se pueden manejar con la **misma referencia**
  - El compilador comprueba que la cabecera a la que se llama es válida
  - **Durante la ejecución se llama al método de acuerdo al tipo real del objeto**
- **El uso del polimorfismo permite**
  - Separación código genérico-abstracto del específico-detalles
  - **Reducción de la cantidad de código**
  - Sustitución de las clases que contienen los detalles
  - **Añadido de nuevas clases con detalles diferentes**

Elementos del polimorfismo

- **Servicio**
  - método que hace algo
- **Servidor**
  - **clase que da el servicio**; define el método, implementa los servicios
- **Interfaz**
  - especificación del servicio pero no da el servicio
  - cabecera del método de servicio
- **Cliente**
  - clase que usa los servicios de otra u otras clases sin saber la clase real del objeto

Clase interfaz

- Contiene **las signaturas de los métodos comunes que definen un servicio**
- Especificación: **clase vacía**, no puede contener
  - **Ni atributos**
  - **Ni constructor**
- Clase incompleta (abstracta) que sólo contiene
  - **cabeceras de métodos públicos (no estáticos)**
  - **constantes**
- Una **interfaz es un tipo**
  - **se pueden declarar referencias**
  - Que podrán apuntar a objetos de cualquier clase que implemente a la interfaz
- **Una clase puede implementar varios interfaces**
  - Si no da cuerpo a todos los métodos: **clase abstracta-incompleta**

Implementación de la interfaz

- Es aquella clase “normal” que hace todo lo que prometía la **interface**, da el servicio:
  - Debe implementar **todos los métodos de la interfaz**
    - **Cabeceras iguales**
    - **Puede añadir excepciones**
  - En la interfaz no se indica quién la implementa
  - En la clase sí se **indica a quién se implementa**
    - class TelefonoMovil **implements Ubicado** {}
- Se pueden implementar **varias interfaces** a la vez

Ejemplos

```
interface Coordenada {
   double getX ();
   double getY ();
   double distancia (Coordenada q);
}
```

```
class Cartesiana
         implements Coordenada {
   private double x, y;
   Cartesiana (double x,double y) {
      this.x=x; this.y=y;
   }
   double getX () { return x; }
   double getY () { return y; }
   double distancia (Coordenada q){
      double dx=q.getX()-this.getX();
      double dy=q.getY()-this.getY();;
      return Math.sqrt(dx*dx+dy*dy);
    }
}
```

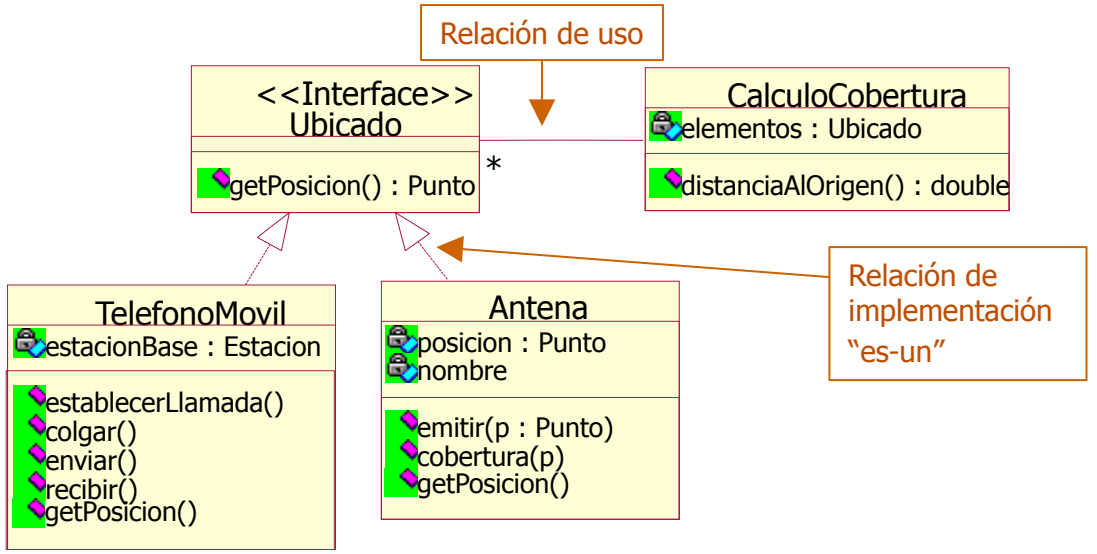
```
class Polar
         implements Coordenada {
   private double r, a;
   Cartesiana (double m,double a) {
      this.r=r; this.a=a;
   }
   double getX() { return r*Math.cos(a); }
   double getY() { return r*Math.sin(a); }
   double distancia (Coordenada q){
      double dx=q.getX()-this.getX();;
      double dy=q.getY()-this.getY();;
      return Math.sqrt(dx*dx+dy*dy);
    }
}
```

Implementación de interfaz

```
interface Ubicado {
  public Punto getPosicion ( );
}

class TelefonoMovil implements Ubicado {
  public Punto getPosicion ( ) {
    Localizador l = new Localizador ( );
    ...
    return l.getCentro ( );
  }
}

class Antena implements Ubicado {
  private Punto posicion;
  public Antena (String nombre, double x, double y) {
    posicion = new Punto (x, y);
  }
  public Punto getPosicion ( ) {
    return posicion;
  }
}
```



Uso del polimorfismo

- **Sólo se pueden crear referencias**
  - Copiarlas y compararlas
  - Dar valor a una referencia
    - Hacer que apunten a un objeto de una clase real que implemente la interfaz
      - ***Ubicado*** *refernenciaDeInterfaz = new* ***ClaseQueImplementaUbicado****();*
  - Se puede llamar **sólo** a los métodos cuyas cabeceras aparecen en la clase interfaz
    - Se ejecutarán los de los objetos reales (polimorfismo)
      - En cada ejecución se mira a qué clase pertenece el objeto real y se ejecuta el método implementado en esa clase
- **No se pueden crear objetos interfaz**
  - **No se puede hacer:** *Ubicado u = new Ubicado();*

Uso del polimorfismo: ejemplo

```
class CalculoCobertura {

    public double distanciaAlOrigen (Ubicado u) {
        Punto p = u.getPosicion ( );
        return Math.sqrt((p.getX() * p.getX()) + (p.getY() * p.getY()));
    }

    public static void main (String [] args) {
        CalculoCobertura c = new CalculoCobertura ( );
        Ubicado u = new TelefonoMovil ( );
        ...

        System.out.println (c.distanciaAlOrigen (u));


        u = new Antena ( );
        ...
        System.out.println (c.distanciaAlOrigen (u));
    }
}
```

u referencia a un objeto TelefonoMovil

u es un **TelefonoMovil.** Durante la ejecución se ejecuta el método getPosicion de la clase **TelefonoMovil**

u referencia a un objeto Antena

u es una **Antena**. Durante la ejecución se ejecuta el método getPosicion de la clase **Antena**

Conversiones

- Generalización (**upcasting**)
  - Tratar a un objeto con una **referencia de tipo más general**
  - Permite usar el objeto de forma **más abstracta**, con menos detalles
  - Sólo se pueden usar los métodos válidos por la referencia
  - El compilador comprueba **los métodos del tipo de la referencia**

```
Ubicado u1 = new TelefonoMovil();
c.distanciaAlOrigen (u1);
System.out.println (u1.getPosicion());
Ubicado u2 = new Antena();
c.distanciaAlOrigen (u2);
System.out.println (u2.getPosicion());
```

Conversiones

- Detallado (**downcasting**)
  - Convertir la referencia a **un tipo más detallado**
  - **Conversión explícita** (forzada)
  - Se pueden llamar a los métodos de la referencia convertida
  - El compilador lo deja en manos del programador
    - Pueden saltar excepciones **ClassCastException**
  - Se puede verificar con el operador: **instanceof()**

```
((TelefonoMovil) u1).establecerLlamada();
((Antena) u2). cobertura (p);
((Antena) u1). cobertura (p);    // ERROR u1 NO es una Antena
                                 // error de ejecución: excepción
if (u1 instanceof Antena) ((Antena) u1).cobertura(p);  //BIEN
```

Jerarquías de interfaces

```
interface  Centrado  {
    public Punto getCentro();
}
interface  Colocable  extends Centrado {
    public void setCentro (Punto p);
}
interface Escalable {
    public void escala (double factor);
}
interface Trasladable  extends Colocable {
    public void mueve (double x, double y);
}
interface Rotable {
    public void rota (double angulo);
}
interface Transformable extends Escalable, Trasladable, Rotable { }
```

Una clase que implemente *Colocable* tendrá: getCentro y setCentro

Una clase que implemente *Trasladable* tendrá: getCentro, setCentro y mueve

Una clase multifunción

- Una misma clase puede implementar varias interfaces

```
interface Mamifero { void amamanta (); }
interface Oviparo { void ponHuevos (); }

class Ornitorrinco implements Mamifero, Oviparo
 { ... }

Ornitorrinco juliana= new Ornitorrinco();
juliana.ponHuevos();
juliana.amamanta();
```

¿Cuándo usar interfaces?

- **Nunca es malo**
  - aunque a veces retarda [un poquitín] la ejecución
- Cuándo se sabe qué queremos; pero
  - no sabemos (aún) cómo hacerlo
  - lo hará otro
  - lo haremos de varias maneras

{PROG} dit UPM

Contenidos

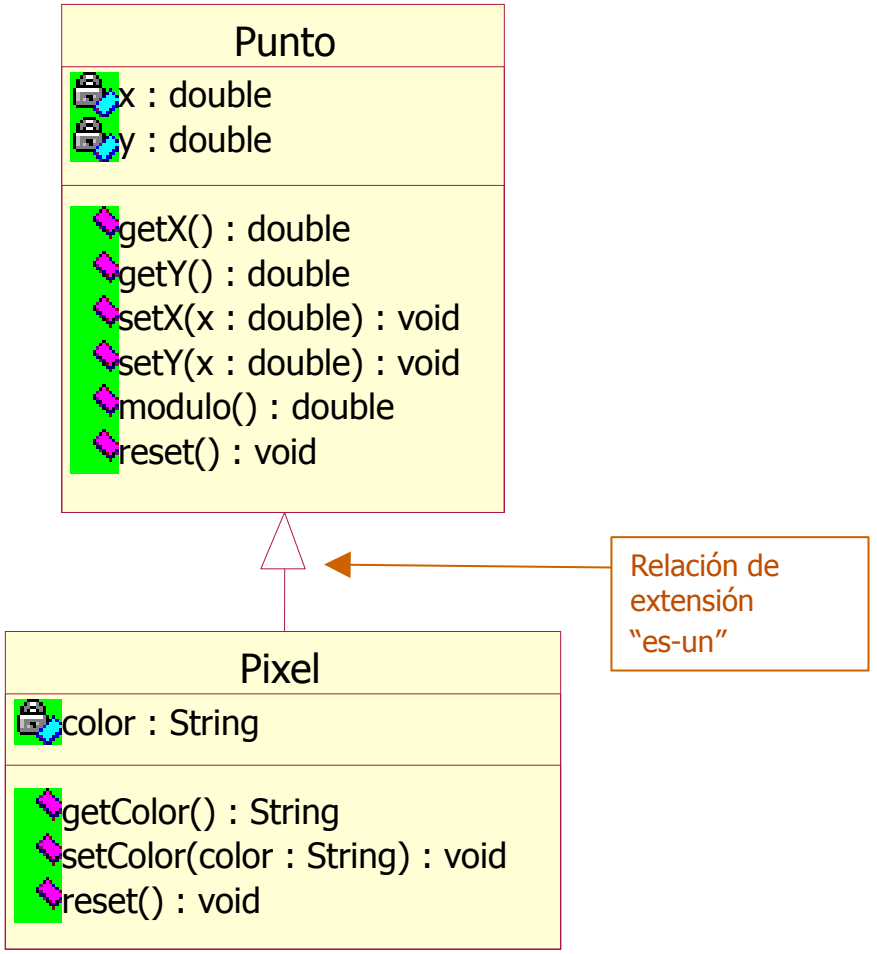
- Polimorfismo
- Implementación de interfaces
- Uso de interfaces
- Jerarquías de interfaces
- **Extensión**
- Jerarquía de extensión

Extensión-herencia

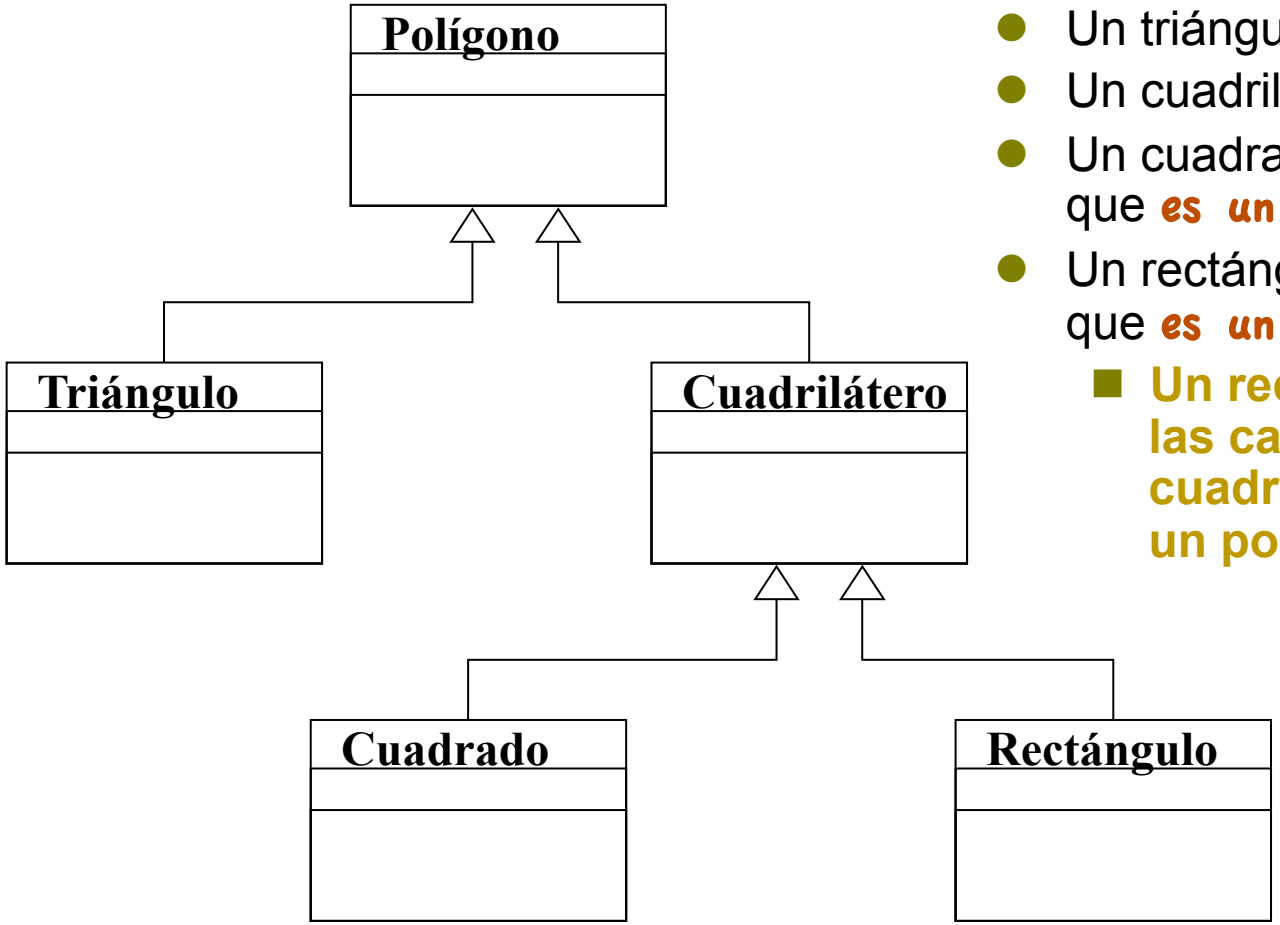
- Herencia: extensión
  - Se parte de una clase que ya está hecha y probada
  - Se crea una **clase nueva que añade atributos y/o métodos**
  - Debe existir alguna relación lógica entre la clase ya hecha (clase base) y la nueva (clase derivada): **es-un**
- El uso de la extensión permite
  - **Reutilizar las clases hechas** como parte de nuevas clases
  - Reducir tiempo y esfuerzo de desarrollo
  - **Aplicar el polimorfismo entre clases** que se parecen porque hay extensión entre ellas
  - **Cambiar el funcionamiento de algún método** de la clase base

Ejemplo

```
class Pixel extends Punto {
    private String color;
    public void setColor (String c) {
        color = c;
    }
    public String getColor() {
        return color;
    }
    public void reset () {
        color = “transparent”;
    }
}
```



Relación de extensión



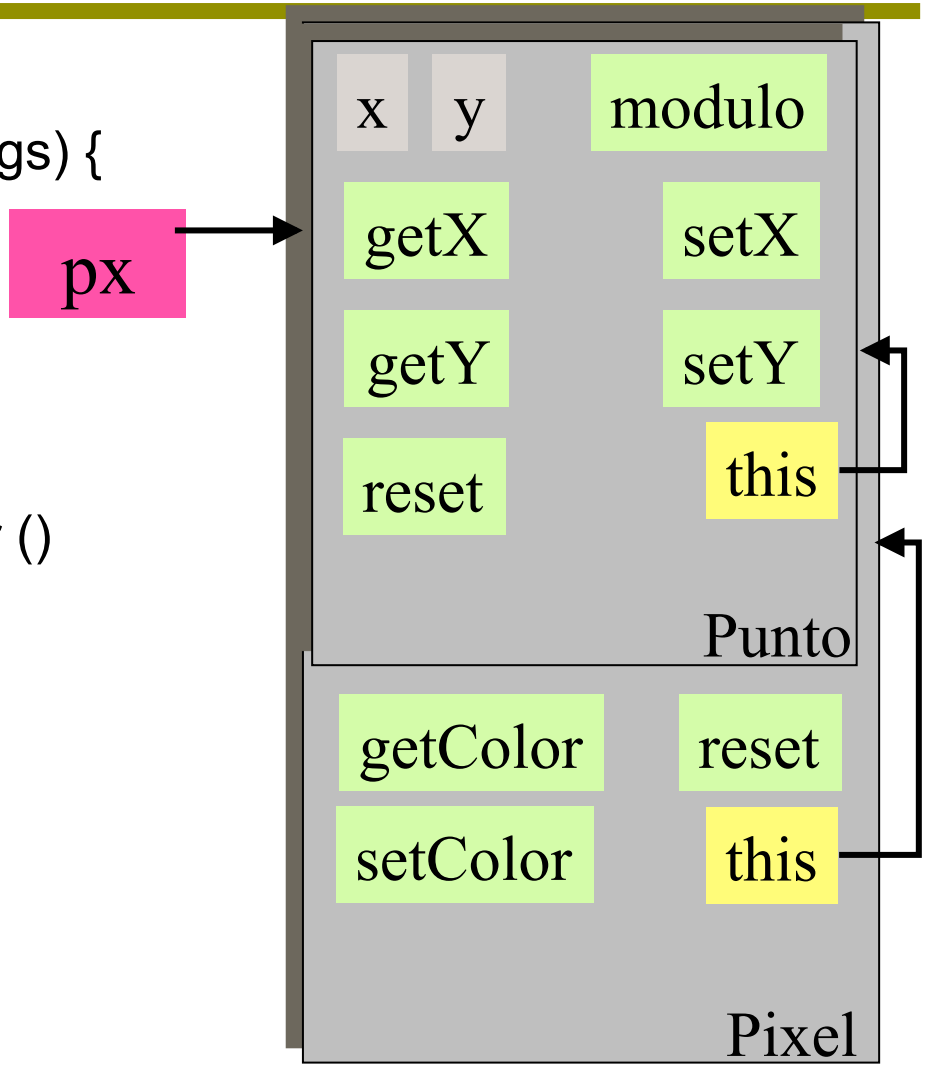
- Un triángulo **es un** polígono
- Un cuadrilátero **es un** polígono
- Un cuadrado **es un** cuadrilátero, que **es un** polígono
- Un rectángulo **es un** cuadrilátero, que **es un** polígono
  - **Un rectángulo tiene todas las características de un cuadrilátero y todas las de un polígono**

Efectos de la herencia

- **Herencia de la interfaz**
  - la interfaz de la clase **derivada contiene la de la clase base**
    - la clase Pixel contiene el método reset, que no lo tiene la clase Punto
    - la clase Pixel contiene todos los métodos públicos de la clase Punto
  - lo que es público en la clase base es como si estuviera en la clase derivada
- **Herencia de la implementación**
  - la implementación de la clase derivada contiene la de la clase base: **todos los atributos y métodos de la clase base incluidos los privados**, aunque no puedan acceder a ellos
  - los objetos de la clase derivada son también de la clase base
    - un Pixel es-un Punto

Ejemplo

```
class PruebaPixel {
  public static void main(String []args) {
    Pixel px = new Pixel ();
    px. setColor (“Green”);
    px. setX(100.0);
    px. setY(45.6);
    System.out.println (px. getColor ()
       + “ “ + px.getX()
       + “ “ + px.getY());
  }
}
```



super

- Referencia automática al objeto interno de la clase base
- Parte de la clase base del objeto
  - "**sube**" un nivel en la jerarquía
  - **permite el acceso a campos y métodos sombreados**
  - permite **construir el objeto de la clase base** llamando a su constructor super()

super.

- En **constructor de la sub-clase**

  `super (parámetros)`

  - **Primera instrucción** del constructor
  - super() automático si no se indica nada

- **En otros métodos** de la sub-clase

```
public String toString () {
        return super.toString() +       //datos de superclase
          "otros datos específicos de la sub-clase");
}
```

  - No tiene que ser la primera instrucción
  - No hay una llamada automática

Construcción con herencia

- Antes de crear un objeto hay que **crear su objeto base**
  - La operación se aplica transitivamente
  - Puede consumir mucho tiempo si hay jerarquía profunda
- Fases de la construcción de un objeto
  1. **Inicializa** todos los campos a cero
  2. Llama a un **constructor de la clase base** (super())

     si se llama a constructor base con parámetros debe ser la primera instrucción
  3. **Ejecuta** el cuerpo del **constructor**

Ejemplo

```
class Punto {
    private double x;
    private double y;
    public Punto (double x, double y) { this.x = x; this.y = y; }
    public Punto () {this(0.0, 0.0); }
}
class Pixel extends Punto {
    public Pixel (float x, float y, Color color) {
        super (x, y);
        this.color= color;
    }
    public Pixel () {this (0.0, 0.0, null); }
}
```

Ámbito de visibilidad

- Los miembros (campos y métodos) de una clase pueden verse…

private

  - Sólo dentro del fichero .java

(package)

  - Sólo dentro del directorio (paquete)

protected

  - Dentro del directorio y en las subclases

public

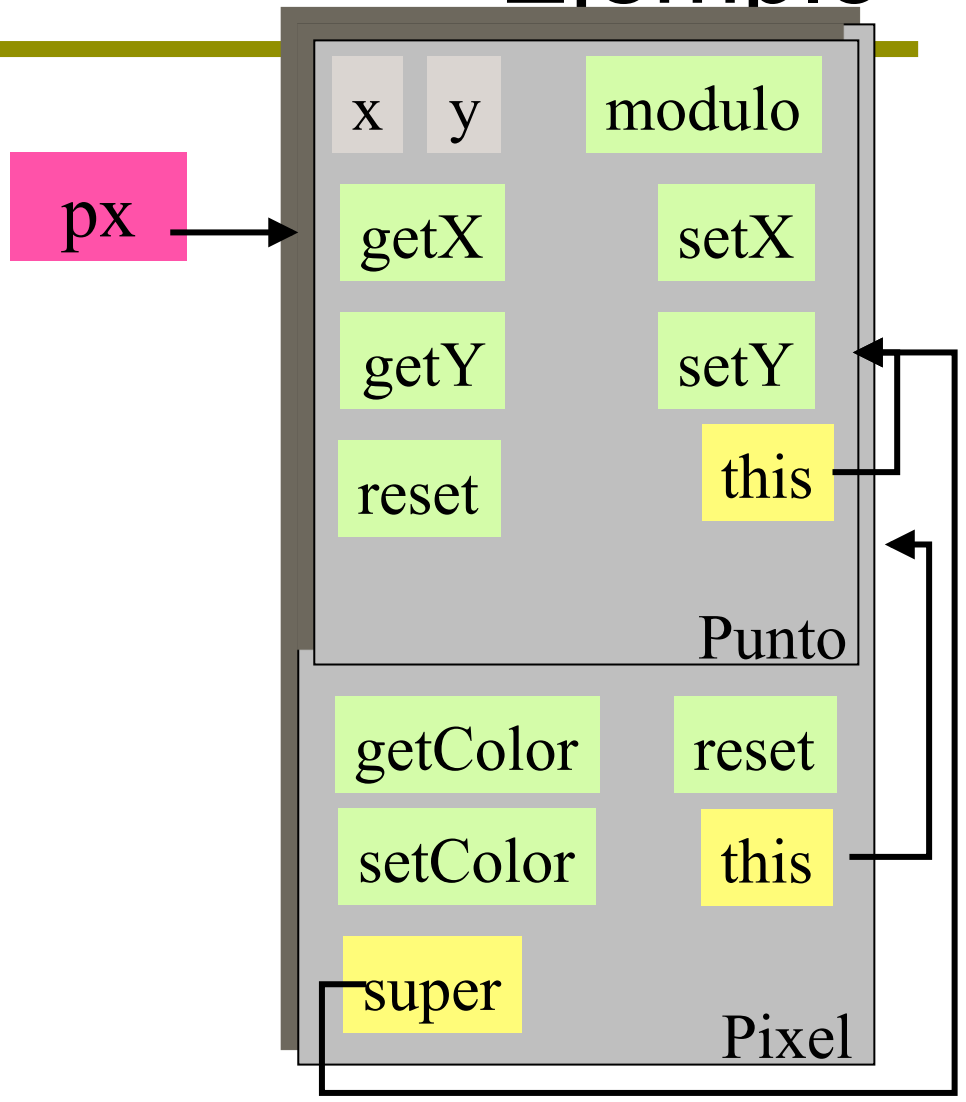
  - Desde cualquier sitio

Sombreado

- **Sombreado**: una clase derivada declara un método público con igual signatura que en la clase base
  - Desde fuera, el de la clase base no se ve
  - Puede hacer más público el de la base
  - Puede lanzar las mismas o menos excepciones
  - El efecto es que cambia el comportamiento del método
- Internamente, se puede ver
  - La referencia super apunta al objeto de clase base
    - Referencia automática privada

Ejemplo

```
class Pixel extends Punto {
   public void reset () {
      super.reset ();
      color =  “transparent”;
   }
   public String toString () {
      return super.toString() + “ “ + color;
   }
}
class PruebaPixel {
  public static void main(String []args) {
    Pixel px = new Pixel ();
    // siempre llama a reset de Pixel
    px. reset ();
  }
}
```



Jerarquía de herencia

- La relación de herencia es transitiva
  - se define una jerarquía de herencia
    - un ingeniero es-un empleado
    - un empleado es-una persona
    - un ingeniero es-una persona
- Todas las clases heredan (directa/ indirectamente) de **Object**
- **Todo objeto deriva de Object**
  - todo objeto es de la clase Object
  - método public String toString(); en Object
  - método public boolean equals (Objetc o); en Object

Transitividad

- La relación de herencia es transitiva
- Se puede hacer ***upcasting***
  - una variable puede referenciar objetos de su propia clase, o de cualquier clase derivada a partir de ella
    - Objetos de su clase y todas sus sub-clases
    - Variables polimórficas
- Se puede hacer ***downcasting***
  - una variable se puede cargar con objetos referenciados por variables de clases antecesoras en la jerarquía de herencia
    - pero la maniobra dará un error si no era un *upcasting*

Conversiones de tipo

- Conversión al tipo base – **upcasting** o generalización
  - Tratar al objeto de clase derivada como si fuera de la clase base
    - El objeto no cambia de tipo, sólo se maneja con una referencia general
    - Sólo se pueden usar los métodos públicos de la clase base
    - Si algún método está sombreado se llama al que sombrea

```
Pixel pixel = new Pixel ();
Punto punto = pixel;
```

- Conversión al tipo derivado – **downcasting** o detallado
  - Indicar que la referencia a clase base apunta a un objeto de derivada
    - Sólo es posible si hubo antes generalización
    - Se puede preguntar si una referencia apunta a un objeto de un tipo

```
if (punto instanceof Pixel) System.out.println ("El pixel es: " + punto.toString());
```

    - Si no, error en ejecución y lanzamiento de excepción (ClassCastException)

```
Pixel pixel = new Pixel();
Punto punto = pixel;
((Pixel)punto).colorea ("Rojo"); // no es seguro
```

{PROG} dit UPM

Clases finales

- Un parámetro **final** no se puede modificar
- Un campo **final** no se puede cambiar
  - es constante
- Un método **final** no se puede sombrear
- Una clase **final** no se puede heredar
- Usos:
  - rendimiento: más rápido
  - seguridad: nadie puede cambiar su comportamiento

Clases abstractas

- **Clases incompletas**
  - Algún método no tiene cuerpo
  - Se indica que se implementa una interfaz pero no se implementan todas las cabeceras de métodos
  - Se parecen a las interfaces, pero pueden tener métodos completos, constructores y atributos
- Uso de las clases abstractas
  - se pueden declarar referencias
  - pero **no crear objetos**
- **Obligan a crear clases derivadas**

Ejemplo

```
abstract class Serie {
   private final int t0;

   protected Serie (int t0) { this.t0 =t0;}

   int t0() {return t0;}

   abstract int termino (int n);

   int suma (int n) {
         int suma = 0;
         for (int i=0; i<n; i++)
                  suma += termino(i);
         return suma;
   }
}
```

polimorfismo

Ejemplo cont.

```
public class SerieAritmetica extends Serie {
   private final int k;

   SerieAritmetica (int a0, int k) { super(a0); this.k=k;}

   int termino (int n){ return t0()+ k*n;}

   int suma (int n) {
        return (t0() + termino(n-1))*n/2;
   }
}
```

```
Serie serie = new SerieAritmetica(1,2);
for (int i=0; i<5; i++)
                System.out.println(serie.termino(i));
```

1 3 5 7 9

Ejemplo cont.

```
public class SerieGeometrica extends Serie {
    private final int k;

    SerieGeometrica (int a0, int k) { super(a0); this.k=k;}

    int termino (int n){
          int t = t0();
          for (int i = 0; i<n; i++) t *=k;
          return t;
    }
    // usamos suma de la clase abstracta
}
```

```
Serie serie = new SerieGeometrica(1,2);
for (int i=0; i<5; i++)
                System.out.println(serie.termino(i));
```

1 2 4 8 16

¿Cuándo usar extensión?

- Pese al tiempo que se le dedica en clase y a lo que de ello hablan los libros, las extensiones se usan poco
- La única razón para extender una clase es poder aplicar *upcasting*
- Si hay dudas entre componer o extender, ¡componga!
  - Razón: los programas son más fáciles de entender
- Si hay duda entre usar un atributo o extender, ¡use el atributo!