Orientación a Objetos en Java

I. Programación ***Basada*** en objetos

II. Programación ***orientada*** a objetos

M. Carmen Fernández Panadero

Raquel M. Crespo García

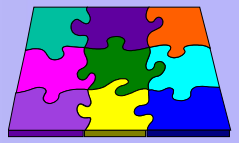
<mcfp, rcrespo@it.uc3m.es>

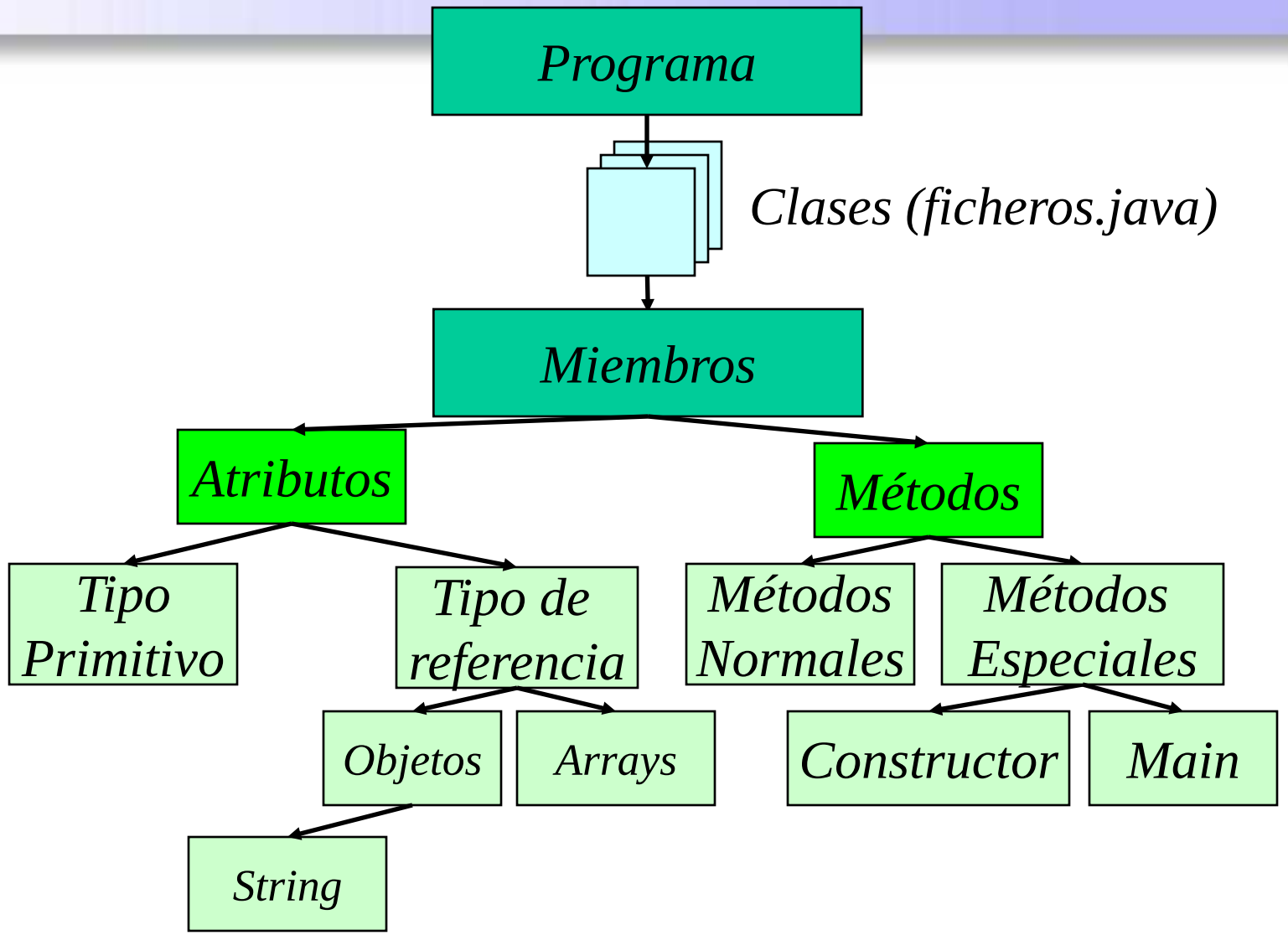
Contenidos

- Polimorfismo
- Ligadura dinámica
- Casting. Compatibilidad de tipos
- Clases y métodos abstractos
  - Implementaciones parciales
  - Polimorfismo con clases abstractas
- Interfaces (concepto e implementación)
  - Herencia múltiple
  - Polimorfismo con interfaces
- Paquetes
- Excepciones

BY–NC–SA

Repaso (Sesión1)





Polimorfismo

Qué es

- Capacidad de un objeto de decidir qué método aplicar, dependiendo de la clase a la que pertenece
  - Una llamada a un método sobre una referencia de un tipo genérico (clase base o interfaz) ejecuta la implementación correspondiente del método dependiendo de la clase del objeto que se creó
- Poli (muchas) + morfo (formas)
  - Una función, diversas implementaciones
- Permite diseñar e implementar sistemas extensibles
  - Los programas pueden procesar objetos genéricos (descritos por referencias de la superclase)
  - El comportamiento concreto depende de las subclases
  - Se pueden añadir nuevas subclases con posterioridad

Polimorfismo
Ejercicio

- Escribid una clase:
  - **`Figura`**, que represente una figura bidimensional (paralelepípedo), con dos atributos para cada dimensión, y un método **`area()`** que calcule el área. Por defecto, esta función devolverá 0
  - **`Triangulo`**, que extienda la anterior y reescriba el método **`area()`**
  - **`Rectangulo`**, que extienda la anterior y reescriba el método **`area()`**
  - **`ListaFiguras`**, que tenga un atributo tipo array de **`Figuras`**, y un método **`areaTotal()`** que devuelva la suma del área de todas las figuras
- ¿Qué tendría que cambiar en **`ListaFiguras`** si añado una nueva clase **`Elipse`**?

Polimorfismo
Ligadura dinámica

- La potencia de la reescritura es que se llama al método correcto, aunque nos estemos refiriendo al objeto de la clase derivada a través de una referencia de la clase base
- Este mecanismo se llama "***ligadura dinámica***"
  - permite detectar **en tiempo de ejecución** cuál es el método adecuado para llamar
- El compilador no genera el código para llamar al método en tiempo de compilación
  - Genera código para calcular qué método llamar

Casting (Conversión de tipos)

Sintaxis y terminología

- Sintaxis:

```
(tipo) identificador
```

- Dos tipos de casting:
  - ***widening***: Una subclase se utiliza como instancia de la superclase. (Ejemplo: llamamos a un método de la clase padre que no ha sido sobreescrito). Es implícito.
  - ***narrowing***: La superclase se utiliza como instancia de una subclase. Conversión explícita.
- Sólo se puede hacer casting entre clases padre e hija no entre clases hermanas

Casting (Conversión de tipos)

Widening o upcasting

1. **Compatibilidad hacia arriba** (*upcasting*)
   - Un objeto de la clase derivada siempre se podrá usar en el lugar de un objeto de la clase base (ya que se cumple la relación "es-un")

```
Persona p = new Alumno();
```

Casting (Conversión de tipos)

Narrowing o downcasting

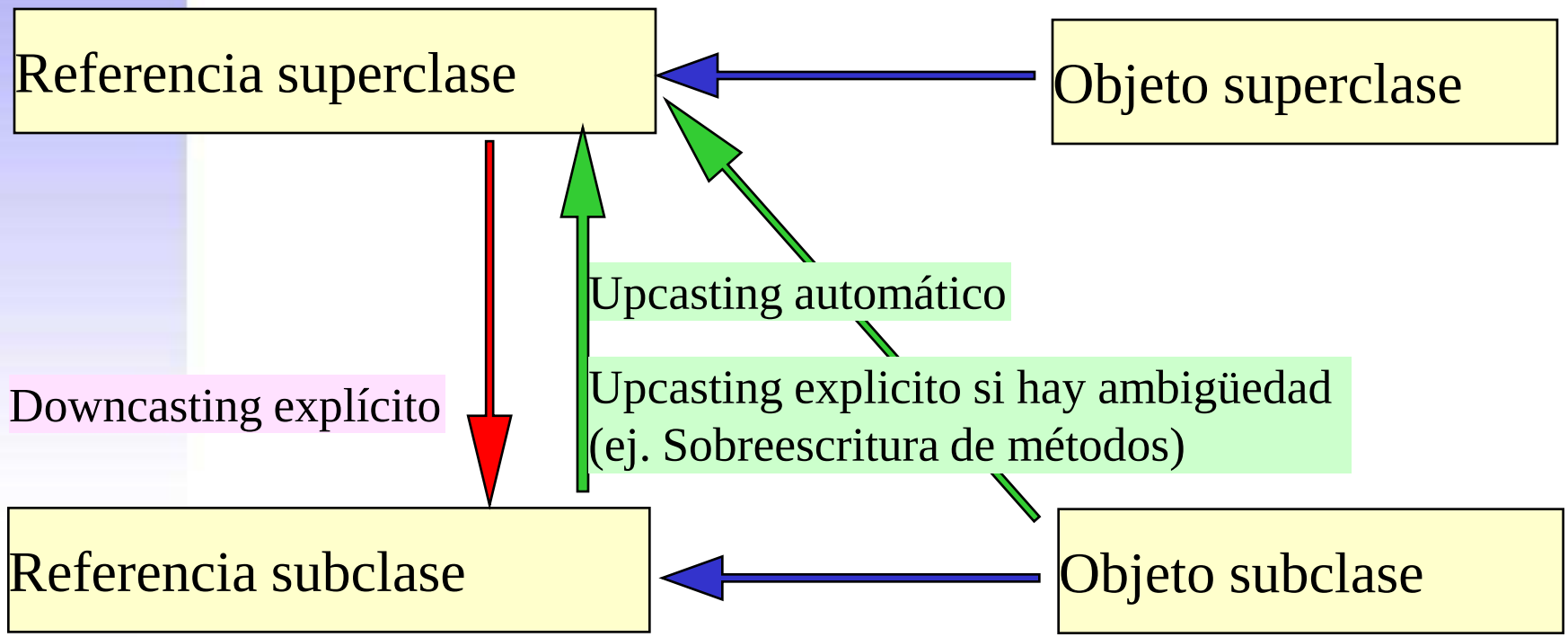
2. **Compatibilidad hacia abajo** (*downcasting*)
   - No se produce por defecto, ya que un objeto de la clase base no siempre es un objeto de la clase derivada

```
Alumno a = new Persona(); // error
```

   - Sólo es posible en los casos en los que el objeto de la clase base realmente sea un objeto de la clase derivada
   - Estos casos se tendrán que indicar explícitamente con un *casting* (con una asignación explícita de la clase).

Casting (Conversión de tipos)

Explícito e implicito



Casting (Conversión de tipos)
Ejemplo

```
public class Prueba2 {
  public static void main (String[] args) {
    Persona p1;
    //conversión ascendente implícita - funciona
    Alumno a1 = new Alumno();
    p1 = a1;

    Alumno a2;
    //conversión descendente implícita - No funciona
    a2 = p1; //error porque no hago conversión explícita

    //conversión descendente explícita - funciona
    a2 = (Alumno) p1;    //p1 referencia una instancia
                         //de Alumno
```

Un alumno siempre es una persona (**implícito**)

Una persona no siempre es un alumno

Si alguien además de persona es alumno (no siempre ocurre) podemos pedirle cosas de alumno pero tendremos que decirle **explicitamente** que le trataremos como alumno.

Casting (Conversión de tipos)
Ejemplo

```
        Persona p2 = new Persona();
        Alumno a3;
    //conversión descendente implícita - no funciona
        a3 = p2;    //da error de compilación

    //conversión descendente explícita - no funciona a veces
    //lanzará la excepción ClassCastException
    //porque p2 no es de la clase Alumno
        a3 = (Alumno) p2; //error


    //conversión descendente implícita - no funciona
        Alumno a4 = new Persona(); //error
    }
}
```

Una persona no siempre es un alumno. No podemos asumir **implicitamente** que lo sea

Una persona no siempre es un alumno. No podemos asumir **implicitamente** que lo sea

Una persona a veces es un alumno pero si no lo es (no lo hemos creado como tal) no podemos tratarlo como si lo fuera, ni siquiera aunque se lo digamos **explicitamente**

Casting (Conversión de tipos)

El operador instanceof

- **`Sintaxis:`**

```
objeto instanceOf clase
```

  - Comprueba si un objeto es realmente de la clase derivada

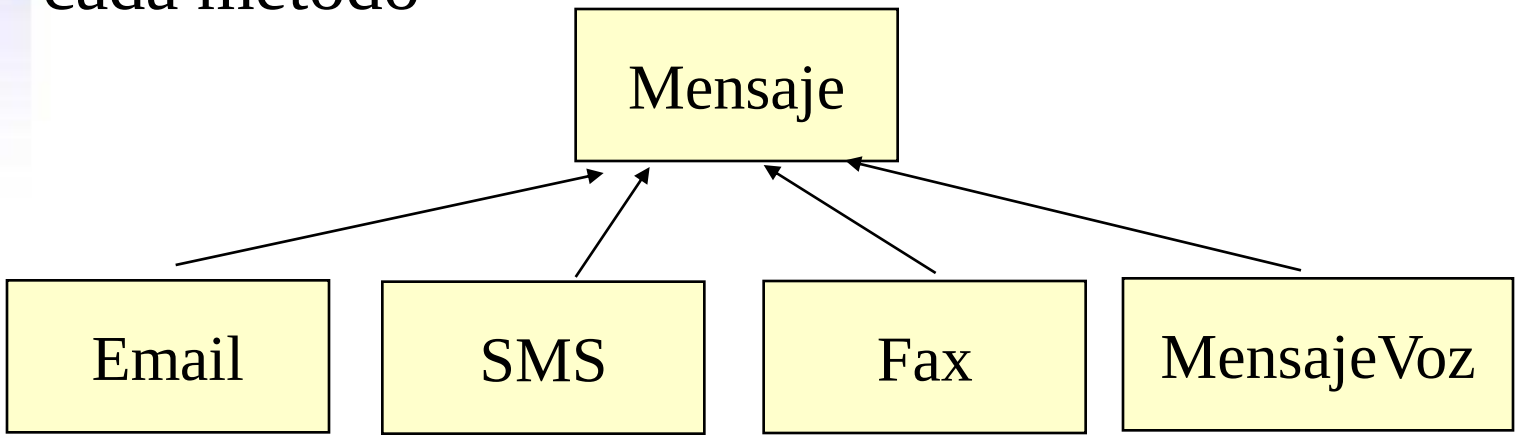
- Ejemplo:

```
  public Alumno comprueba (Persona p) {
  Alumno a = null;
  if (p instanceOf Alumno)
    a = (Alumno) p;
  return a;
}
```

Clases abstractas

¿Qué son?

- Aquellas que tienen ***al menos un método abstracto*** (sin implementar, sin código).
- Declara la ***estructura*** de una determinada ***abstracción***, sin implementar completamente cada método



Clases abstractas

Características

- Las clases y métodos abstractos se definen con la palabra clave ***abstract***

```
public abstract class Figura {...}
```

- No pueden llevar el modificador abstract:
  - los **constructores**
  - los métodos **estáticos**
  - los métodos **privados**

Clases abstractas

Características

- ***No podemos crear objetos*** de una clase abstracta
  - Pueden existir referencias a clases abstractas
  - Pero apuntarán a objetos de clases derivadas de la clase abstracta.

```
Figura fig = new Rectangulo(2,3);
```

- ***Sí podemos heredar*** de una clase abstracta
- En una clase abstracta puede haber
  - Metodos **abstractos**
  - Métodos **no abstractos**

Clases abstractas

¿Para qué sirven?: Implementaciones parciales

- Las clases abstractas suelen usarse para representar clases con ***implementaciones parciales***
  - Algunos métodos no están implementados pero sí declarados
- El objetivo de las implementaciones parciales es dar una ***interfaz común*** a todas las clases derivadas de una clase base abstracta
  - Incluso en los casos en los que la clase base no tiene la suficiente información como para implementar el método

Clases abstractas

Métodos abstractos:

- Métodos declarados pero no implementados en las clases abstractas

```
abstract tipoResultado nombre (listaParametros);
```

  - Se declaran con la palabra reservada **`abstract`**
- Las clases que hereden de la clase abstracta deberán implementar los métodos abstractos de la superclase
  - O serán abstractas ellas también

NOTA: **No hay llaves**!! No están implementados después de la declaración se pone solo un **;**

Clases abstractas

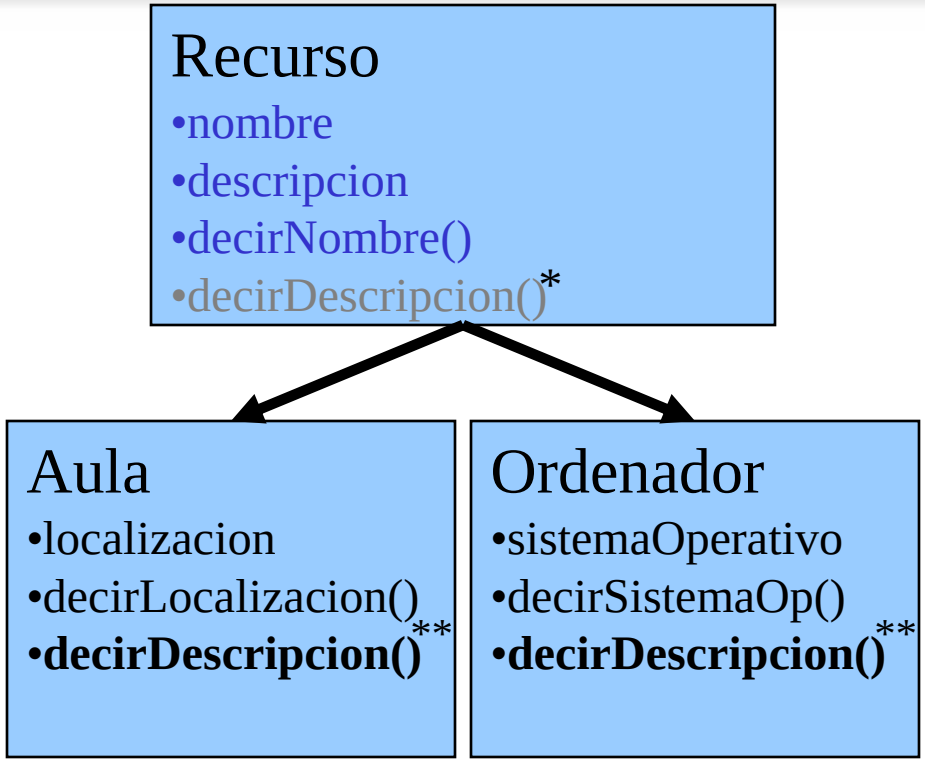
¿Cómo se usan?

La clase Recurso es ***abstracta*** *porque* ***uno de sus métodos*** *decirDescripcion()* ***no tiene código***

* El color gris claro indica que no tiene código

Todas las clases que hereden *de recurso* ***tienen que tener*** *un método decirDescripcion() donde pongamos* ***el código***

** La "negrita" indica que tiene código




```
public abstract class Recurso
public class Aula extends Recurso
public class Ordenador extends Recurso
```

Clases abstractas

¿Cómo se usan?. Ejemplo

```
abstract class Figura {
   double dim1;
   double dim2;

    Figura(double dim1, double dim2){
      this.dim1 = dim1;
      this.dim2 = dim2;
   }

    abstract double area();
}
```

```
class Rectangulo extends Figura {
    Rectangulo(double dim1, double dim2){
      super(dim1,dim2);
   }
    double area(){
      // area de Rectangulo
      return dim1*dim2;
   }
}
```

Clases abstractas

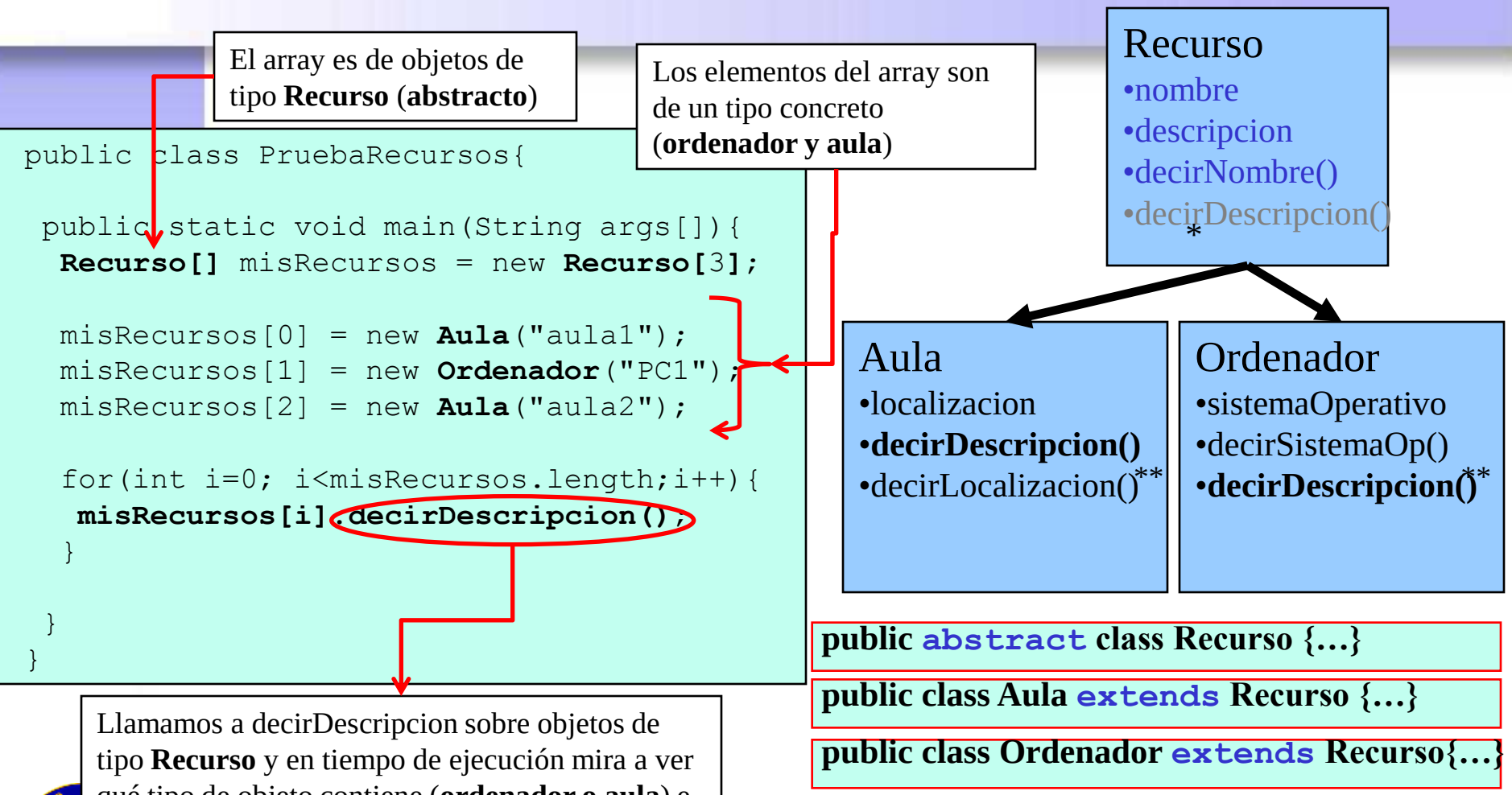
Polimorfismo

El array es de objetos de tipo **Recurso** (**abstracto**)

Los elementos del array son de un tipo concreto (**ordenador y aula**)

```
public class PruebaRecursos{

 public static void main(String args[]){
  Recurso[] misRecursos = new Recurso[3];

  misRecursos[0] = new Aula("aula1");
  misRecursos[1] = new Ordenador("PC1");
  misRecursos[2] = new Aula("aula2");

  for(int i=0; i<misRecursos.length;i++){
   misRecursos[i].decirDescripcion();
  }

 }
}
```

Llamamos a decirDescripcion sobre objetos de tipo **Recurso** y en tiempo de ejecución mira a ver qué tipo de objeto contiene (**ordenador o aula**) e invoca el método correspondiente esto es lo que llamamos **Ligadura dinámica**



* El color gris claro indica que no tiene código

** La "negrita" indica que tiene código

Interfaces

¿Qué son?

- Los interfaces llevan el concepto de clase abstracta un paso más adelante
  - ***Todos* los métodos de un interfaz son abstractos**
  - Un interfaz pueden considerarse "similar" a una clase abstracta "pura"
- El acceso a un interfaz es ***público***
  - Los atributos en los interfaces son implícitamente public, static y final
  - Los métodos en los interfaces no tienen modificadores de acceso, son públicos
- Los interfaces son ***implementados*** por clases
  - una ***clase*** implementa un interfaz definiendo los cuerpos de ***todos*** los métodos.
  - una ***clase abstracta*** implementa un interfaz rellenando el cuerpo o bien declarando abstractos los métodos
  - una clase puede implementar uno o más interfaces (herencia múltiple)

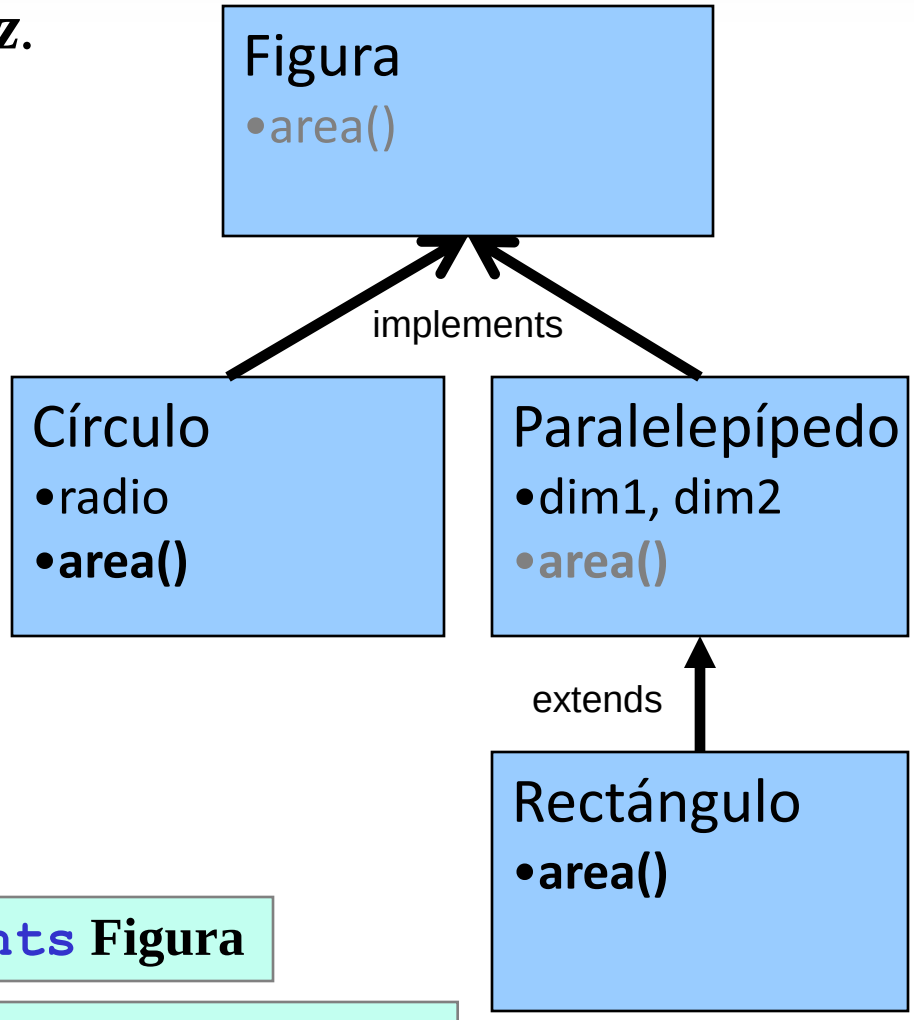
Interfaces
¿Qué son?

- Una ***interfaz*** es un elemento puramente de **diseño**
  - ¿**Qué** se quiere hacer?
- Una ***clase*** (incluidas las abstractas) es una mezcla de **diseño e implementación**
  - ¿**Qué** se quiere hacer y **cómo** se hace?
- Suponen una abstracción completa de una clase
  - Abstrae sus características y comportamientos públicos de su implementación (el modo en el que se ejecutan esos comportamientos)
- Distintas clases pueden implementar la interfaz de distintas formas

Interfaces

¿Cómo se usan?

*Figura no una clase, es una **interfaz**. Sólo define el comportamiento, pero no la implementación.*



*Todas las clases que implementen Figura tienen que implementar **todos** los métodos declarados en Figura (o declararlos abstractos)*

```
public interface Figura
```

```
public class Circulo implements Figura
```

```
public class Paralelepipedo implements Figura
```

Interfaces
Declaración

- Sintaxis:

```
visibilidad interface nombre {
  tipo variable = valor;
  tipoDevuelto nombreMetodo(listaParametros);
}
```

  - *Visibilidad* es **`public`** o se omite
  - Todos los métodos son implícitamente **`abstract`** y **`public`**
  - Las variables de la interfaz son **`static`** y **`final`**
    - Representan constantes

NOTA: **No hay llaves**!! No está implementado después de la declaración se pone sólo un **;**

Interfaces
Implementación

- Si una clase implementa una interfaz, quiere decir que implementa todos los métodos abstractos de esa interfaz
- Esto se representa con la palabra reservada **`implements`**:

```
class ClaseDerivada extends ClaseBase
    implements Interfaz1, Interfaz2 {...}
```

Interfaces

¿Cómo se usan?. Ejemplo

- Definid una interfaz para objetos que puedan ser imprimibles
  - Método void imprime()
- Haced que las clases Rectangulo e Email implementen dicha interfaz

Interfaces

¿Cómo se usan?. Ejemplo

```
interface Imprimible {
   void imprime();
}
```

```
class Email extends Mensaje
             implements Imprimible{

 public void imprime(){
    System.out.println("Imprimiendo email");
    System.out.println(mensaje);
 }
```

NOTA: **No hay llaves**!! No está implementado después de la declaración se pone sólo un **;**

Interfaces

¿Cómo se usan?. Ejemplo

```
class Rectangulo extends Figura implements Imprimible{
   [...]
 public void imprime(){
    System.out.println("Imprimiendo Rectangulo (" + dim1 + " x " + dim2 + ")");
      StringBuffer res = new StringBuffer();
      for (int i = 0; i <= dim1+1; i++)
         res.append("* ");
      res.append("\n");
      for (int j = 0; j < dim2; j++){
         res.append("* ");
         for (int i = 1; i <= dim1; i++)
            res.append("  ");
         res.append("*");
         res.append("\n");
      }
      for (int i = 0; i <= dim1+1; i++)
         res.append("* ");

      System.out.println(res);
   }
}
```

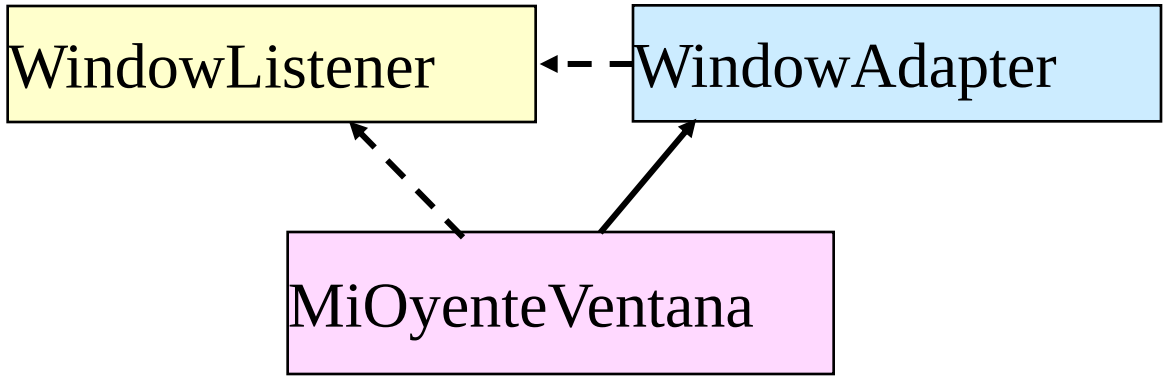
Interfaces

¿Cómo se usan? Extensión de interfaces.

- Las interfaces también pueden extenderse (heredarse) unas de otras
- En ese caso, se van acumulando los métodos que deberán incluir las clases que implementen las interfaces

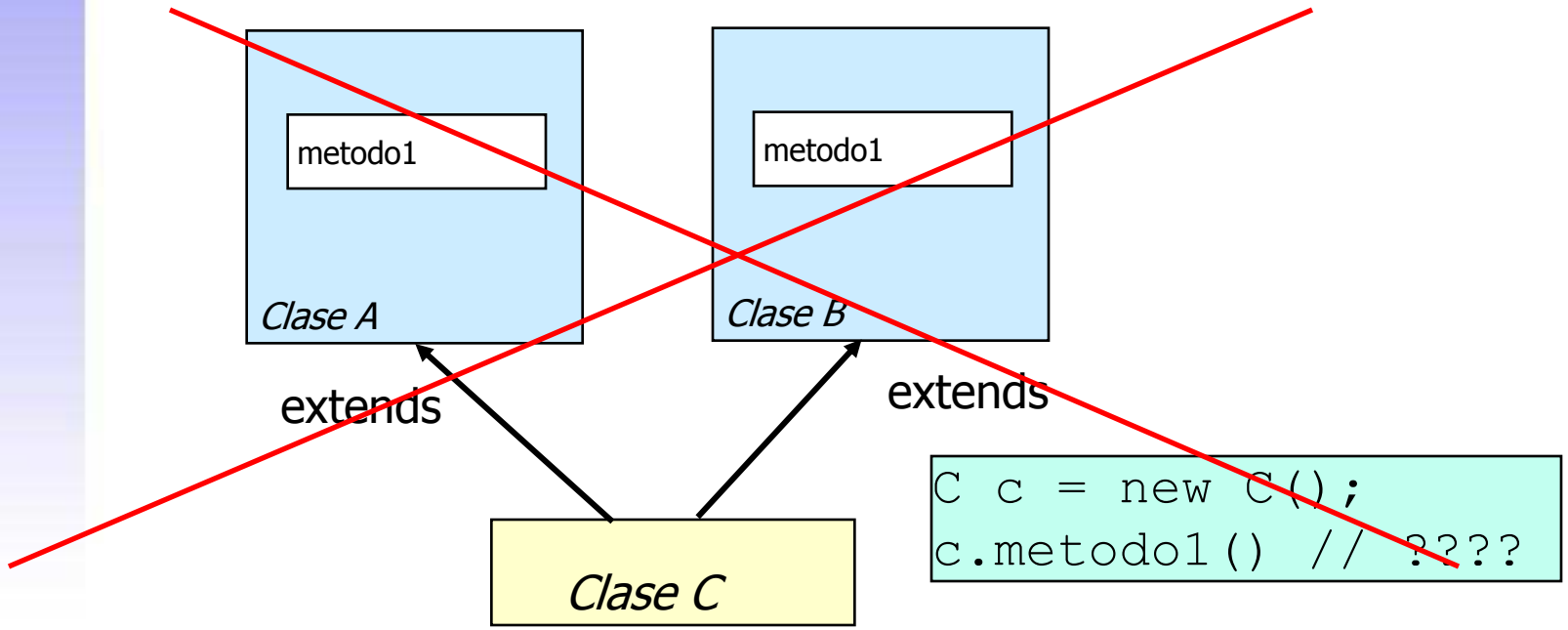
Interfaces
¿Cómo se usan?. Ejemplo

- Un buen programa siempre trabajará con **interfaces** y **extensiones** de clases
- En el futuro, los programadores pueden decidir si ampliarlo:
  - **extendiendo** la implementación o
  - **implementando** la interfaz



Interfaces

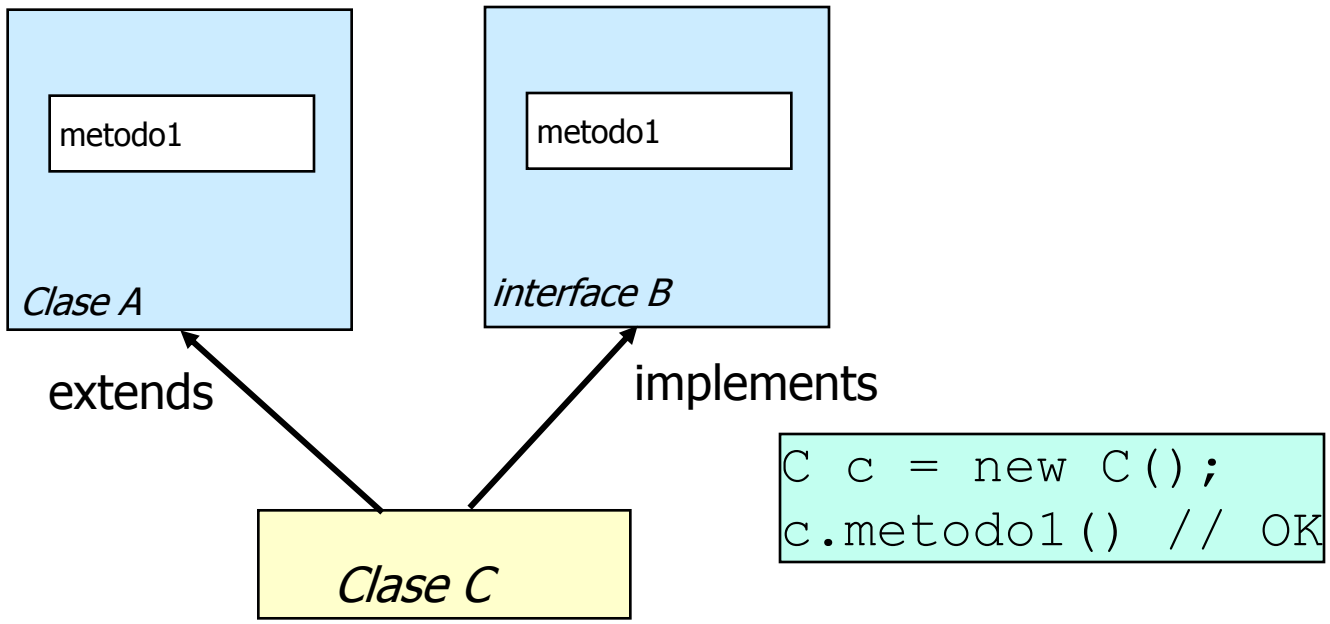
¿Para qué sirven? Herencia múltiple



- En Java **no** existe la herencia múltiple
- Funcionalidad similar gracias a las **interfaces**

Interfaces

¿Para qué sirven? Herencia múltiple

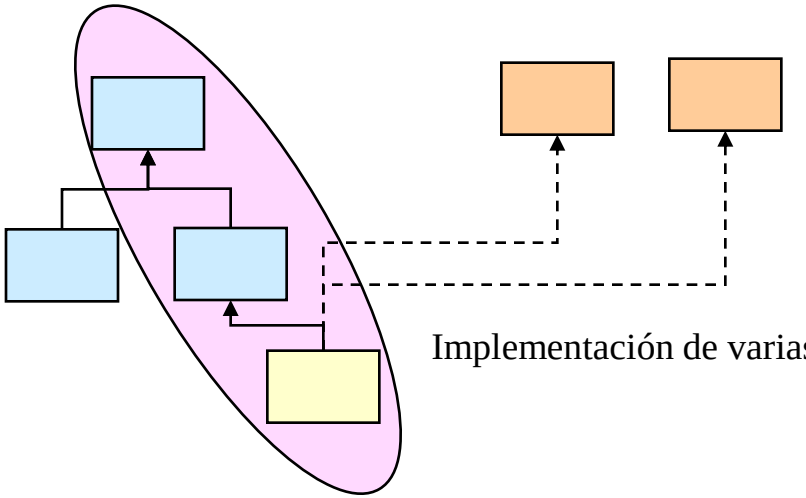


- Una clase hereda de una única superclase
- Pero puede implementar varios interfaces

Interfaces

¿Para qué sirven? Herencia múltiple

- ***Herencia simple*** de **implementaciones**
  - Extensión de una sola clase
- ***Herencia múltiple*** de **interfaces**
  - Implementación de varias interfaces



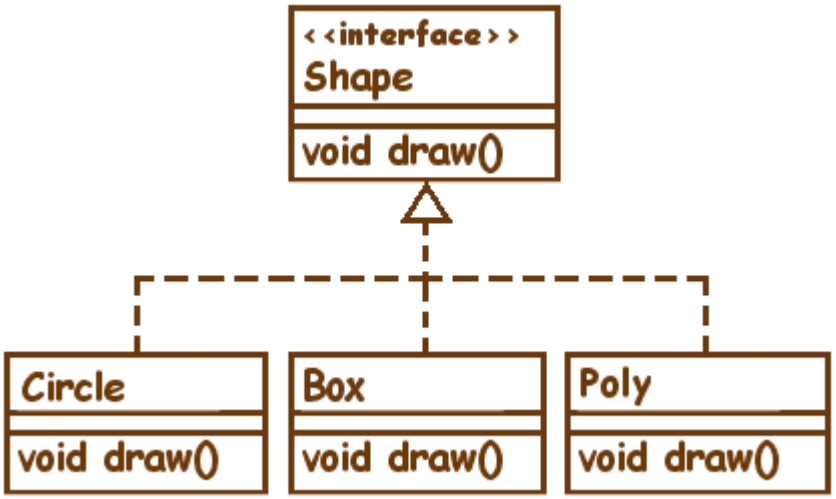
Implementación de varias interfaces

Interfaces

¿Para qué sirven? Polimorfismo

- Polimorfismo: “una interfaz, múltiples métodos”
- Las interfaces dan soporte a la resolución dinámica de métodos durante la ejecución (ligadura dinámica)
- ¿Qué diferencia hay entre la implementación de interfaces y la herencia?
  - Las interfaces no forman parte de la jerarquía de herencia

Ejercicio: JavaRanch



Ejercicio: JavaRanch

```
import java.awt.* ;
public interface Shape
{
    public void draw( Graphics g );
}
```

```
import java.awt.* ;
public class Circle implements Shape
{
    private int x ;
    private int y ;
    private int wide ;
    private int high ;
    private Color color ;

    Circle( int x , int y , int wide , int high , Color color )
    {
        this.x = x ;
        this.y = y ;
        this.wide = wide ;
        this.high = high ;
        this.color = color ;
    }

    public void draw( Graphics g )
    {
        g.setColor( color );
        g.fillOval( x , y , wide , high );
    }

}
```

Ejercicio: JavaRanch

```
      import java.awt.* ;
public class Box implements Shape
{
        private int x ;
        private int y ;
        private int wide ;
        private int high ;
        private Color color ;
        Box( int x , int y , int wide , int high , Color color )
        {
            this.x = x ;
            this.y = y ;
            this.wide = wide ;
            this.high = high ;
            this.color = color ;
        }

        public void draw( Graphics g )
        {
            g.setColor( color );
            g.fillRect( x , y , wide , high );
        }
    }
```

Ejercicio: JavaRanch

```
import java.awt.* ;
public class Poly implements Shape
{
    int[] x ;
    int[] y ;
    private Color color ;

    Poly( int[] x , int[] y , Color color )
    {
        this.x = x ;
        this.y = y ;
        this.color = color ;
    }

    public void draw( Graphics g )
    {
        g.setColor( color );
        g.fillPolygon( x , y , x.length );
    }
}
```

Ejercicio: JavaRanch

```
import java.awt.* ;
public class ShowShapes extends Frame
{
    static int[] vx = { 200 , 220 , 240 , 260 , 280 , 250 , 230 };
    static int[] vy = { 150 , 150 , 190 , 150 , 150 , 210 , 210 };

    static Shape[] shapes =
    {
        // J
        new Box( 50 , 70 , 100 , 20 , Color.red ) ,
        new Box( 90 , 70 , 20 , 110 , Color.blue ) ,
        new Circle( 50 , 150 , 60 , 60 , Color.green ) ,
        new Circle( 70 , 170 , 20 , 20 , Color.white ) ,
        new Box( 50 , 90 , 40 , 90 , Color.white ) ,

        // a
        new Circle( 130 , 150 , 60 , 60 , Color.green ) ,
        new Box( 170 , 180 , 20 , 30 , Color.blue ) ,
        new Circle( 150 , 170 , 20 , 20 , Color.white ) ,
```

Ejercicio: JavaRanch

```
            // v
            new Poly( vx , vy , Color.black ) ,

            // a
            new Circle( 290 , 150 , 60 , 60 , Color.green ) ,
            new Box( 330 , 180 , 20 , 30 , Color.blue ) ,
            new Circle( 310 , 170 , 20 , 20 , Color.white ) ,
        };

        ShowShapes()
        {
            setBounds( 200 ,150 , 400 , 250 );
            setVisible( true );
        }

        public void paint( Graphics g )
        {
            for( int i = 0 ; i < shapes.length ; i++ )
            {
                shapes[ i ].draw( g );
            }
        }

        public static void main( String[] args )
        {
            new ShowShapes();
        }

    }
```

Resumen Orientación a objetos

- ***Clase*** (concreta)
  - ***Todos*** los métodos implementados
- ***Clase abstracta***
  - ***Al menos un*** método no implementado, sólo declarado
  - modificador `abstract`
- ***Interfaz***
  - ***Nada*** de implementación
  - palabra reservada: `interface`

Resumen Orientación a Objetos

- ***Clase*** (concreta o abstracta)
  - puede ***extender*** (`extends`) a ***una*** sola clase (herencia simple)
  - puede ***implementar*** (`implements`) ***uno o más*** interfaces (herencia múltiple)
    - palabra reservada: `extends`
- ***Interfaz***
  - puede extender (`extends`) a ***uno o más*** interfaces

Paquetes
(Packages)

M. Carmen Fernández Panadero
Raquel M. Crespo García
<mcfp, rcrespo@it.uc3m.es>

Paquetes

- Un ***paquete*** agrupa ***clases*** e ***interfaces***
- Las jerarquías de un paquete se corresponden con las jerarquías de directorios
- Para referirse a miembros y clases de un paquete se utiliza la notación de separarlos por puntos.
  - Ej: Cuando hacemos un applet importamos la clase Applet que nos proporciona Java

```
import java.applet.Applet;
```

  - La clase java.applet.Applet está en el directorio java/applet

Paquetes

- ***¿Cómo utilizar paquetes creados por otros?***
  - Incluimos en el classpath la dirección de la carpeta que contiene el paquete. Por ej: suponiendo que PaqueteDeOtro está en las carpetas c:\java\lib (windows) y /opt/lib (linux):

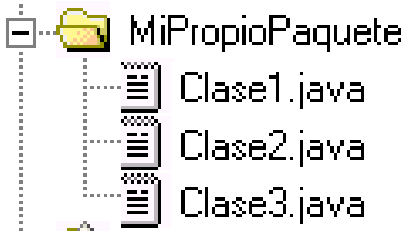
```
set CLASSPATH=c:\java\lib;%CLASSPATH%          (windows)
setenv CLASSPATH /opt/lib/:$CLASSPATH          (linux)
```

  - En la clase que queramos utilizarlo ponemos antes de la declaración de la clase la sentencia import correspondiente

```
import PaqueteDeOtro.*;
```

- ***¿Cómo crear mis propios paquetes?***
  - Almaceno mis clases en un directorio con el nombre del paquete
  - Pongo al principio de todas las clases que pertenezcan al paquete la instrucción

```
package MiPropioPaquete;
```



MODIFICADORES		*clase*	*metodo*	*atributo*
acceso	public	Accesible desde cualquier otra clase		
	(friendly)	Accesible sólo desde clases de su propio paquete		
	protected		Accesible desde la clase y sus subclases	
	private		Accesibles sólo dentro de la clase	
otros	abstract	No se pueden instanciar Son **para heredar** de ellas Al menos 1 método abstracto	No tiene código Se implementa en las subclases o clases hijas	
	final	No se puede heredar de ellas. Es la hoja en el árbol de herencia	No se puede ocultar Es **cte** y no puede ser modificado en las clases hijas	No se puede cambiar su valor, es **cte**. Se suele utilizar en combinación con static
	static	Clase de nivel máximo. Se aplica a classes internas	Es el mismo para todos los objetos de la clase. Se utiliza: NombreClase.metodo();	Es la misma para todos los objetos de la clase.

BY-NC-SA

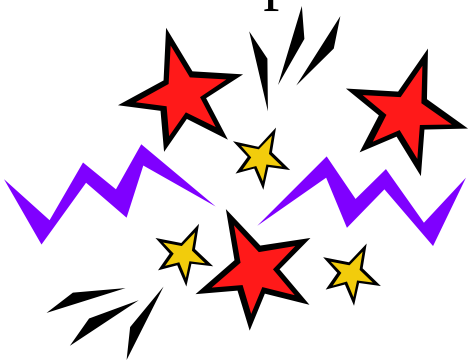
Excepciones

M. Carmen Fernández Panadero
Raquel M. Crespo García
<mcfp, rcrespo@it.uc3m.es>

Excepciones

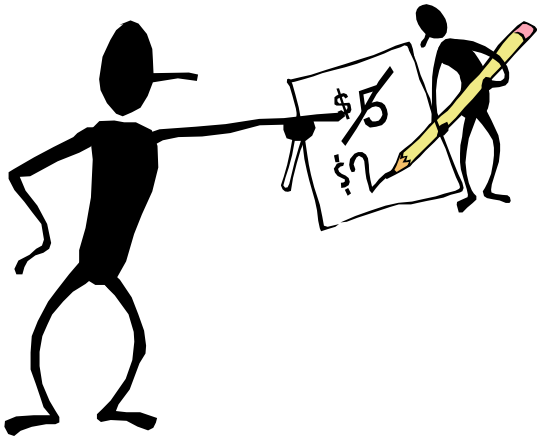
- Qué son
- Para qué sirven
- Qué tipos hay
- Cómo utilizarlas

Excepcion



Ignorar ➡ Terminar

Tratar



Lanzar

Excepciones: ¿Qué son?

- **Evento** que al ocurrir impide la ejecución normal del programa.
- Cuando ocurre se crea un **objeto excepción** y se pasa al sistema de **control de ejecución**
- El sistema de control de ejecución:
  - **Busca** un trozo de código que maneje la excepción
  - Si no lo encuentra **termina** el programa

Excepciones: ¿Para qué sirven?

- Para **separar** el tratamiento de excepciones del código normal (`try-catch`)
- Para **propagar** errores en la pila de llamadas (`throws`)
- Para **agrupar** y diferenciar tipos de errores (al ser objetos pueden agruparse por clases)
- Cualquier método tiene que:
  - **tratar** (catch) o
  - **pasar** (throws)

  cualquier excepción producida durante su ejecución

Excepciones: ¿Qué tipos hay?

- Hay fundamentalmente 2 tipos:
  - De tiempo de **ejecución** (`RuntimeException`)
    - No se comprueban al compilar
    - Ej: ArithmeticException, NumberFormatException, IndexOutOfBoundException, NullPointerException, etc.)
  - El resto de excepciones se comprueban en tiempo de compilación
    - Ejemplo: de **entrada y salida** (`IOException, FileNotFoundException, EOFException`)
    - Definidas por el **usuario** (`MyException`)
- En tiempo de compilación se comprueba que todas las excepciones (excepto las de tiempo de ejecución):
  - se **capturen** o
  - se **declaren** en los métodos en los que puedan darse

Excepciones: ¿Cómo usarlas?

- Cómo se produce:
  - implicitamente (cuando se produce un error)
  - explícitamente `throw` new IOException(mensaje)
- Qué hacer:
  - **Tratarla**:
    - Rodear con `try{}` las sentencias susceptibles de lanzar excepciones
    - Rodear con `catch(nombreException){}` las sentencias que se deben ejecutar cuando se produce
  - **Lanzarla** public void miMetodo `throws` IOException
- Rodear con `finally{}` el código que queremos que se ejecute siempre