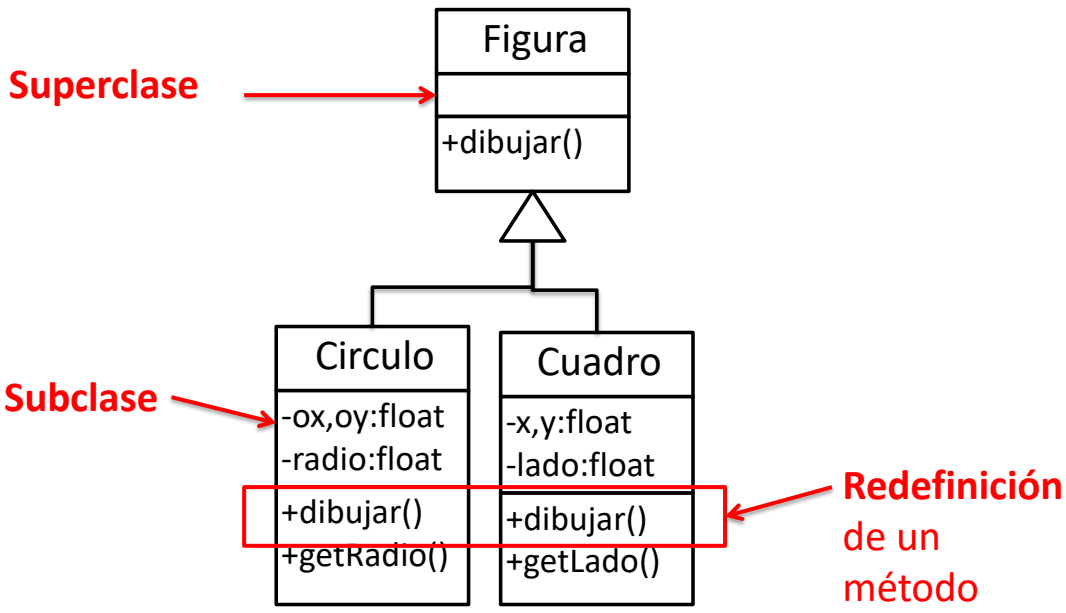
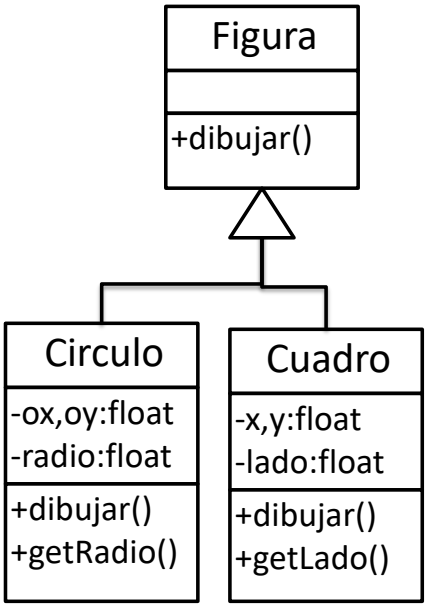
Polimorfismo

Programació Orientada a Objectes

Algunos Conceptos Previos (I)



Algunos Conceptos Previos (II)



- Referencias a objetos de una misma clase

```
Circulo ci = new Circulo();
ci.getRadio();
Cuadro cu = new Cuadro();
cu.getLado();
```

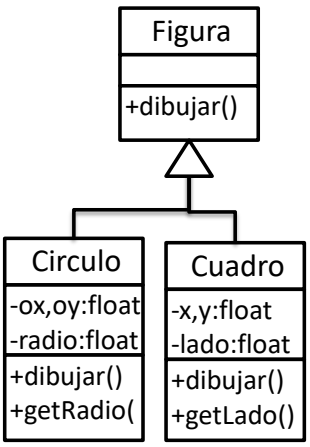
- Referencias de superclase a objetos de una subclase.

```
Figura f = new Circulo();
f.dibujar();
```

~~`f.getRadio();`~~ ←Error de compilación!

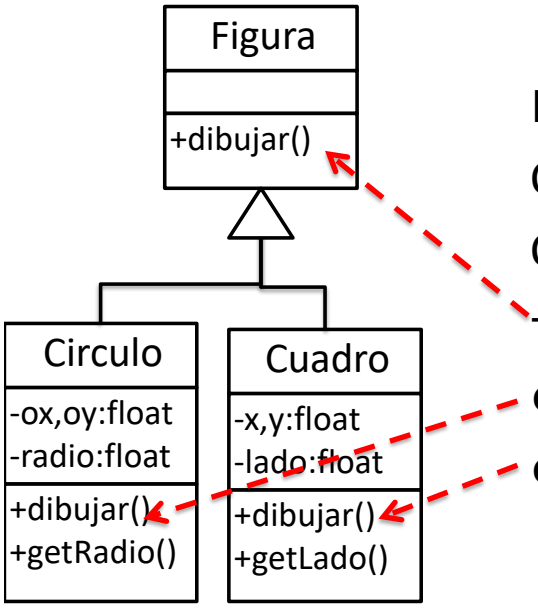
Algunos Conceptos Previos (III)

- Cuando nos referimos a una subclase a través de una referencia a su superclase, podemos llamar a las funciones de la subclase haciendo un ***Class Casting***:



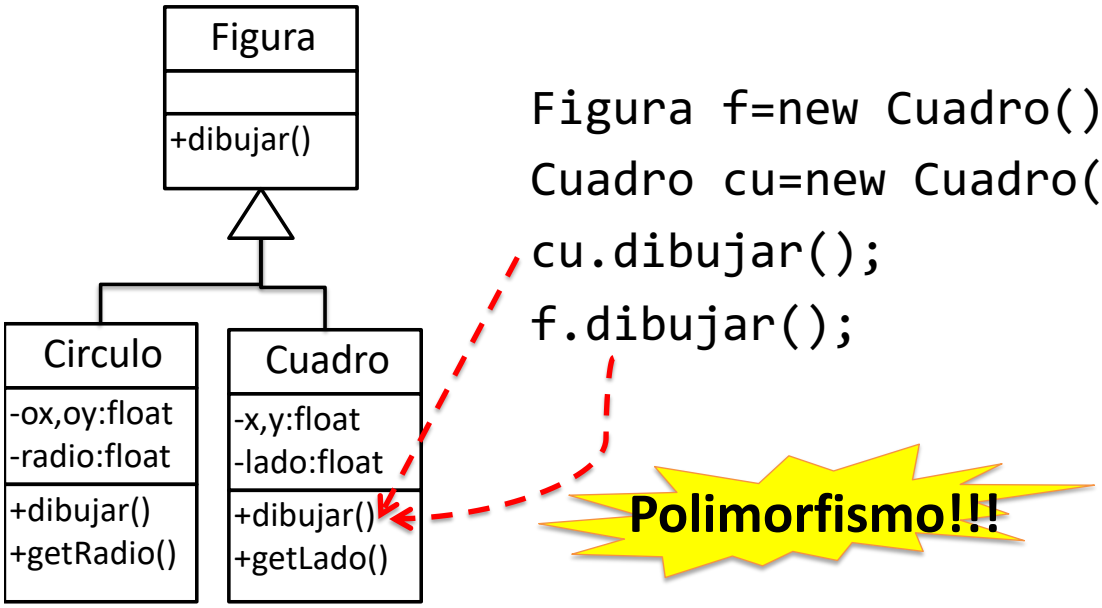

```
Figura f = new Circulo();
f.dibujar(); //OK
Circulo c = (Circulo) f;
c.getRadio();
/*OJO!! El compilador se «tragará» lo siguiente, pero al ejecutar nos saltará un error de Class Cast*/
Cuadro cu = (Cuadro) f;
```

¿Qué métodos se llaman en cada momento? (I)




```
Figura f=new Figura();
Circulo ci=new Circulo();
Cuadro cu=new Cuadro();
f.dibujar();
ci.dibujar();
cu.dibujar();
```

¿Qué métodos se llaman en cada momento? (II)




```
Figura f=new Cuadro();
Cuadro cu=new Cuadro();
cu.dibujar();
f.dibujar();
```

Polimorfismo

- Es la habilidad de cambiar el comportamiento de una variable dependiendo del tipo de objeto al que hace referencia.
- Permite que múltiples objetos de diferentes subclases puedan ser tratados como objetos de su superclase, mientras que automática y transparentemente se ejecutarán los métodos que se sobrecarguen en la subclase.
- Considerado el **tercer pilar** de la POO, junto con encapsulación y herencia.

Ejemplo 1: toString()

•Todas las clases de java derivan (aunque no se especifique con el "extends") de la clase Object, que define, entre otros, el método toString()

```
public class Complex {
  private double r;
  private double i;
 (... constructores y otros
    métodos...)
}
```

```
public class ComplexPolimorfico {
  private double r;
  private double i;
  (... constructores y otros métodos...)
  public String toString() {
    return r + " + " + i + "i";
  }}
```

•Observad la salida de este código:

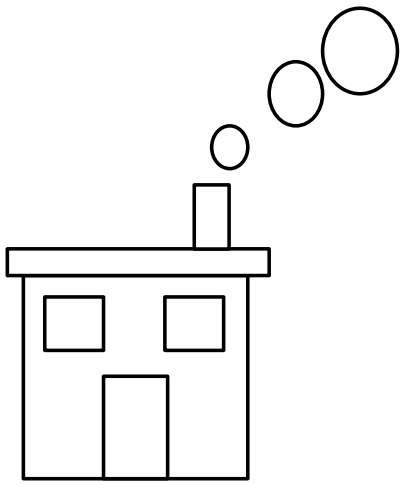
```
Object c1 = new Complex(1,2);
Object c2 = new ComplexPolimorfico(1,2);
System.out.println("c1 = " + c1.toString());
System.out.println("c2 = " + c2.toString());
```

•Salida:

```
c1 = poo.paquete.Complex@732dacd1
c2 = 1 + 2i
```

Ejemplo 2: programa de dibujo

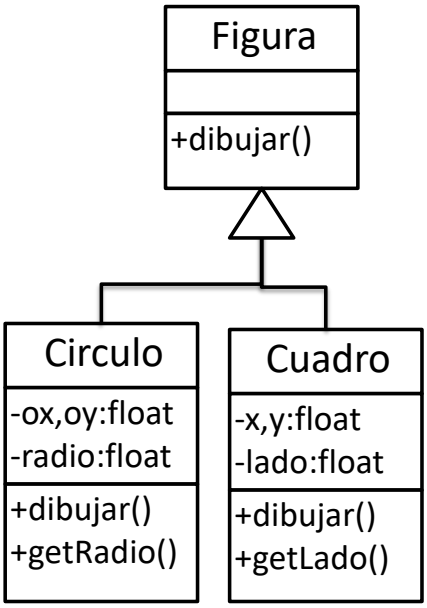
```
public class PlanoDibujo {
  List<Figura> figuras = new ArrayList<Figura>();

  public void agregaFigura(Figura f) {
    figuras.add(f);
  }

  public void dibujarPlano() {
    for(Figura f : figuras) {
      f.dibuja();
    }
  }
}
```



Beneficios del polimorfismo

- Simplicidad
  - Un código que necesita manejar una familia de clases puede ignorar los detalles específicos de cada subclase y simplemente interactuar con la superclase.
  - El código es más sencillo tanto para el que lo escribe, como para que los demás lo entiendan.
- Extensibilidad
  - No costaría nada añadir nuevas subclases a la familia (ej: Triangulo), y éstas funcionarían sin problemas en el código ya existente.

¿Qué debería dibujar el siguiente código?




```
Figura ci=new Circulo();
ci.dibujar();
Figura cu=new Cuadro();
cu.dibujar();
Figura f=new Figura();
f.dibujar();
```

¿¿¿Cómo podemos dibujar algo «abstracto»???

Clases abstractas

- Definen algunos métodos y atributos.
- Contienen métodos «abstractos»
  - Métodos que solo definen una cabecera, pero no contienen código en su interior.
- Las subclases de una clase abstracta deben implementar de manera **obligatoria** los métodos abstractos de su superclase.
  - (A no ser que las subclases también sean abstractas)
- Una clase abstracta no se puede instanciar
  - Pero sí se pueden usar referencias a una clase abstracta.

Ejemplo de clase abstracta: Zoo

```
public abstract class Animal {

  protected String especie;
  public void setEspecie(String especie) {
     this. especie = especie;
  }
  public String getEspecie() {
     return especie;
  }

  public abstract void saludar();
}
```

Ejemplo de clase abstracta: Zoo

```
public class Gato extends Animal {
  public void saludar() {
     System.out.println(“miau!”);
}}
public class Perro extends Animal {
  public void saludar() {
     System.out.println(“guau!”);
}}
public class Pato extends Animal {
  public void saludar() {
     System.out.println(“cuac!”);
}}
```

Ejemplo clase abstracta: Zoo

```
Perro perro = new Perro(); //OK
Gato gato = new Gato(); //OK
Animal animal = new Animal(); /* MAL: una
        clase abstracta no se puede instanciar */
Animal animal = new Pato(); /* OK: sí se pueden
                        usar referencias del tipo
                    de una clase abstracta para
                    referirse a sus subclases */
```

Interfaces

- Definen un conjunto de métodos vacíos que describen una funcionalidad concreta.
  - Son un "contrato" entre la clase que la usa y la clase que la implementa
- No contienen atributos, ni código.
  - Aunque sí pueden definir constantes
- Se diferencia de las «clases» en:
  - No establecen una jerarquía «padre-hijo», sino que agrupan clases por ciertas funcionalidades
  - Una clase puede ***implementar*** varias interfaces, mientras que **sólo puede tener una superclase**.
- Al igual que con las clases abstractas, pueden usarse referencias a interfaces, pero no instanciarse.

Ejemplo: enchufes (I)

```
public interface Enchufable {
   // Implementa el funcionamiento de cualquier
   // electrodoméstico enchufable a la red eléctrica
   void funcionar();
}
public interface TomaCorriente {
    // Conecta un objeto enchufable a la toma de corriente
   // Si e == null, es que no hay nada enchufado
   void enchufar(Enchufable e);

   // Cuando haya corriente, llamará al método
   // "funcionar" del objeto enchufado
   void proporcionaCorriente();
}
```

Ejemplo: enchufes (II)

```
public class TomaPared implements TomaCorriente {
   private Enchufable enchufable = null;

   public void enchufar(Enchufable enchufable) {
      this.enchufable = enchufable;
   }
   public void proporcionaCorriente() {
      if(enchufable != null)
          enchufable.funcionar();
   }
}
public class Televisor implements Enchufable {
   public void funcionar() {
         System.out.println("Mostrando tu serie favorita");
   }
}
```

Ejemplo: enchufes (III)

```
TomaCorriente toma = new TomaPared();
Televisor tv = new Televisor();
```

También se puede instanciar como:
Clavija tv = new Televisor();

```
toma.proporcionaCorriente();
```

No pasa nada

```
toma.enchufar(tv);
toma.proporcionaCorriente();
```

TV está
"Mostrando tu serie favorita"

Ejemplo: Enchufes (V)

```
public class MaquinillaAfeitar implements Clavija {
   public void funcionar() {
         System.out.println("Bzzzzzzzzz!");
   }
}

TomaCorriente toma = new TomaPared();
Clavija tv = new Televisor();
Clavija maquinilla = new MaquinillaAfeitar();

toma.enchufar(tv);
toma.proporcionaCorriente();
```

"Mostrando tu serie favorita"

```
toma.enchufar(maquinilla);
toma.proporcionaCorriente();
```

"Bzzzzzzzzz!"

Ejemplo: enchufes (VI)

```
public class TomaInterruptor implements TomaCorriente {

    private Clavija clavija = null;
    private boolean activado = false;

    public void setActivado(boolean activado) {
        this.activado = activado;
    }
    public void enchufar(Clavija clavija) {
        this.clavija = clavija;
    }
    public void proporcionaCorriente() {
        if(activado && clavija != null)
                clavija.funcionar();
    }
}
```

Ejemplo: enchufes (y VII)

```
TomaInterruptor toma = new TomaInterruptor();

MaquinillaAfeitar maquinilla =
                                new MaquinillaAfeitar();

toma.enchufar(maquinilla);
toma.proporcionaCorriente();

toma.setActivado(true);
toma.proporcionaCorriente();
```

No pasa nada

"Bzzzzzzzzzzz!"

Ventajas del polimorfismo/interfaces

- Podemos crear nuevas clases que implementen `Clavija` (p. ej. `Radio`, `VideoConsola`, …) y enchufarlas en cualquier toma de corriente, sin modificar las clases ya existentes

- Podemos crear nuevas clases que implementen `TomaCorriente` (p. ej. `TomaConTemporizador`) y enchufar cualquier `Clavija` existente sin tener que modificar el funcionamiento de éstas.

- El polimorfismo permite extender nuestro software sin tener que modificar/cambiar software ya existente
  - Menos trabajo
  - Menos errores

Interfaces que ya conocíamos: contenedores

- Un contenedor de un mismo tipo puede estar guardado en memoria de muchas maneras.
- Independientemente de la estructura interna en memoria, facilita las cosas el poder usar un tipo de contenedor siempre de la misma manera → Usar siempre referencias de tipo interfaz

```
List<Alumno> lista1 = new ArrayList<Alumno> ();

List<Producto> lista2 = new LinkedList<Producto>();

Map<String,Alumno> mapa1 = new HashMap<String,Alumno>();

Map<String,Producto> mapa2 = new TreeMap<String,Producto>();

~~List<Alumno> lista3 = new List<Alumno>();~~ //error!

~~Map<String,Producto> map3 = new Map<String,Producto>();~~ //error!
```

Ejemplo de Interfaz en Java: Comparable<T>

- **TreeSet** y **TreeMap** guardan los datos según un orden natural. Es necesario que los objetos que se guardan en TreeSet y las claves de un TreeMap implementen la interfaz "Comparable"
  - (Ver javadoc de Comparable)

```
public  class Alumno implements Comparable<Alumno>{
  private String nombre, DNI;
  private int edad;
  public int compareTo(Alumno other) {
     return this.DNI.compareTo(other.DNI); // string es Comparable
  }}
public class Producto implements Comparable<Producto> {
   private float precio;
   private String nombre;
   public int compareTo(Producto other) {
      return Float.compare(precio, other.precio);
   }}
```

A destacar: los dos objetos no tienen propiedades comunes (sería incorrecto que pertenecieran a la misma familia), pero sí tienen operaciones en común (tiene sentido que compartan interfaz).

Herencia/Implementación múltiple

- Una clase solamente puede tener una superclase, pero puede implementar varias interfaces.

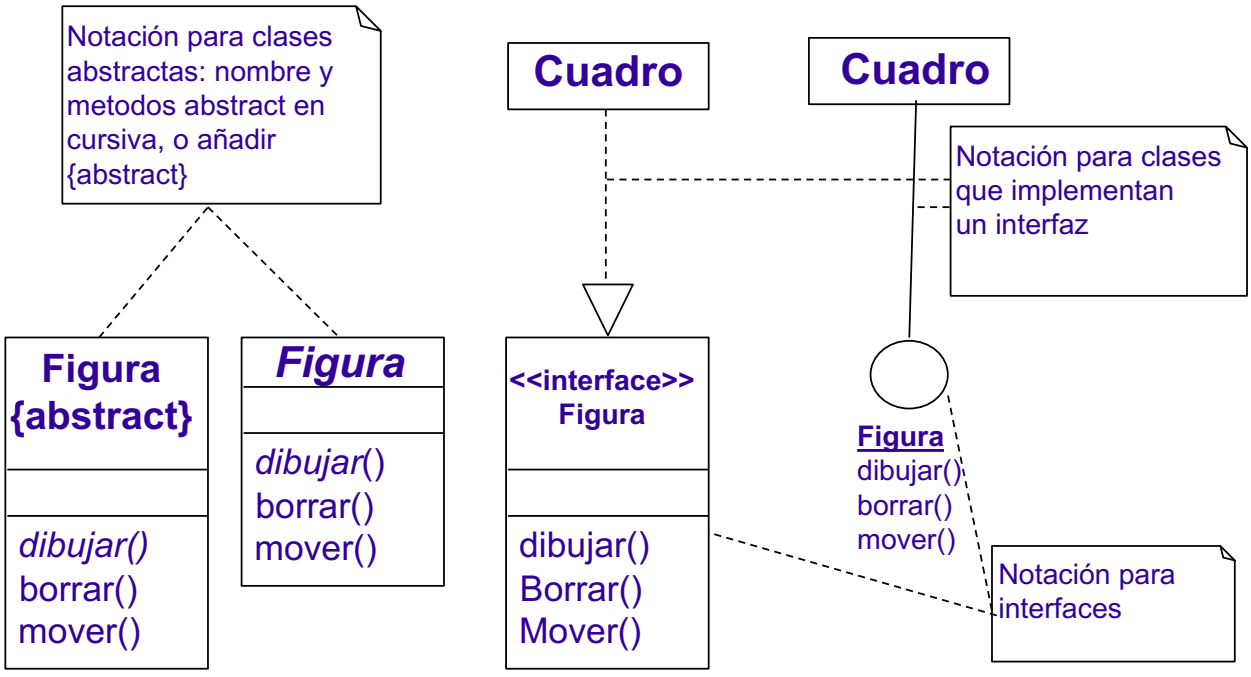
Ejemplo:

```
public class Silla extends Mueble
    implements  Sentable, Desmontable, Apilable
{
 /* código */
}
```

Tabla comparativa

Tipo de clase base	Clase corriente	Abstracta	Interfaz
Contiene atributos y código	Sí	Sí	No (sólo constantes)
Pueden crearse referencias de su tipo	Sí	Sí	Sí
Se puede instanciar	Sí	No	No
Herencia/implementación múltiple	No	No	Sí
Obligaciones de las subclases/clases implementadoras	Ninguna	Obliga a implementar los métodos abstractos de la superclase	Obliga a implementar todos los métodos de la interfaz

Notación UML Diagrama de Clases



Herencia/polimorfismo: palabras clave

- **super**
  - Referencia a los métodos/constructores/atributos a una clase base.

```
public class Superclase {
   public Superclase(String valor){ (…) }
   public String toString() { return valor; }
}
public class Subclase extends Superclase {
   public Subclase() { super("un valor cualquiera"); }
   public String toString() {
      return "Superclase.toString = " + super.toString();
   }
}
```

Herencia/polimorfismo: palabras clave

- **instanceof**
  - Operador para saber a qué tipo pertenece una instancia.

```
Figura obj = iterator.next();
if(obj instanceof Circulo) {
  Circulo ci = (Circulo) obj;
  System.out.println("radio = "+ci.getRadio());
} else if(obj instanceof Cuadro) {
  Cuadro cu = (Cuadro) obj;
  System.out.println("lado = "+cu.getLado());
}
```

Ejercicio

- Dadas las interfaces TomaCorriente y Clavija de los ejemplos anteriores, crear la clase **Regleta**
  - Se enchufa a una toma de corriente (implementa Enchufable)
  - Se le pueden enchufar 3 clavijas (implementa TomaCorriente)
    - El método "enchufar" enchufa una clavija en alguna de las 3 tomas libres (lo decide la clase)
    - Si se llama a "enchufar" cuando están las 3 clavijas ocupadas, se desconecta la que lleva más tiempo enchufada
  - Sólo proporciona corriente si está enchufada a otra toma, y ésta le proporciona corriente

Ejercicio para rizar el rizo

- Hacer la clase **RegletaInterruptor**
- Intentad no reescribir el código que ya hayáis escrito en la clase "Regleta"