Parte I: Programación en un lenguaje orientado a objetos

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Estructuras algorítmicas
4. Datos compuestos
5. Tratamiento de errores
6. Entrada/salida con ficheros
7. *Herencia y Polimorfismo*

- Herencia. Clases Abstractas. Polimorfismo.

Jerarquía de clases

Uno de los mecanismos de la programación orientada a objetos es poder crear clases a partir de otras, por extensión

- Cuando la nueva clase se parece a la anterior se programan solo las extensiones, sin repetir lo común
  - programación por *extensión*

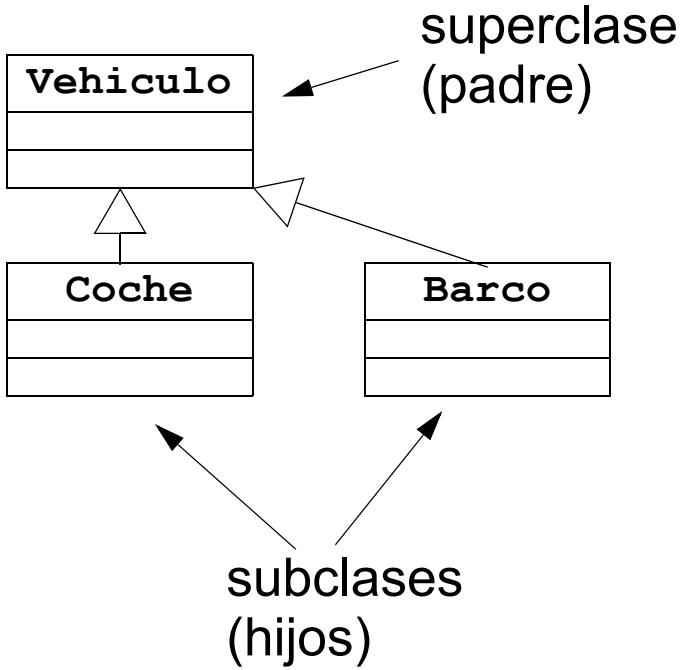
Esto da lugar a una jerarquía:

- clase padre o ***superclase***
- clases hijas o ***subclases***

Herencia

Ejemplo de relación de ***herencia***:

- todos los coches son vehículos
- pero no al revés



La ***herencia***: mecanismo para crear nuevas clases a partir de otras existentes,

- heredando, y posiblemente redefiniendo, y/o añadiendo ***operaciones***
- heredando y posiblemente añadiendo ***atributos***

El mecanismo de herencia ***no suprime*** abributos ni operaciones

Herencia de operaciones

Al extender una clase

- se ***heredan*** todas las operaciones del padre
- se puede ***añadir*** nuevas operaciones

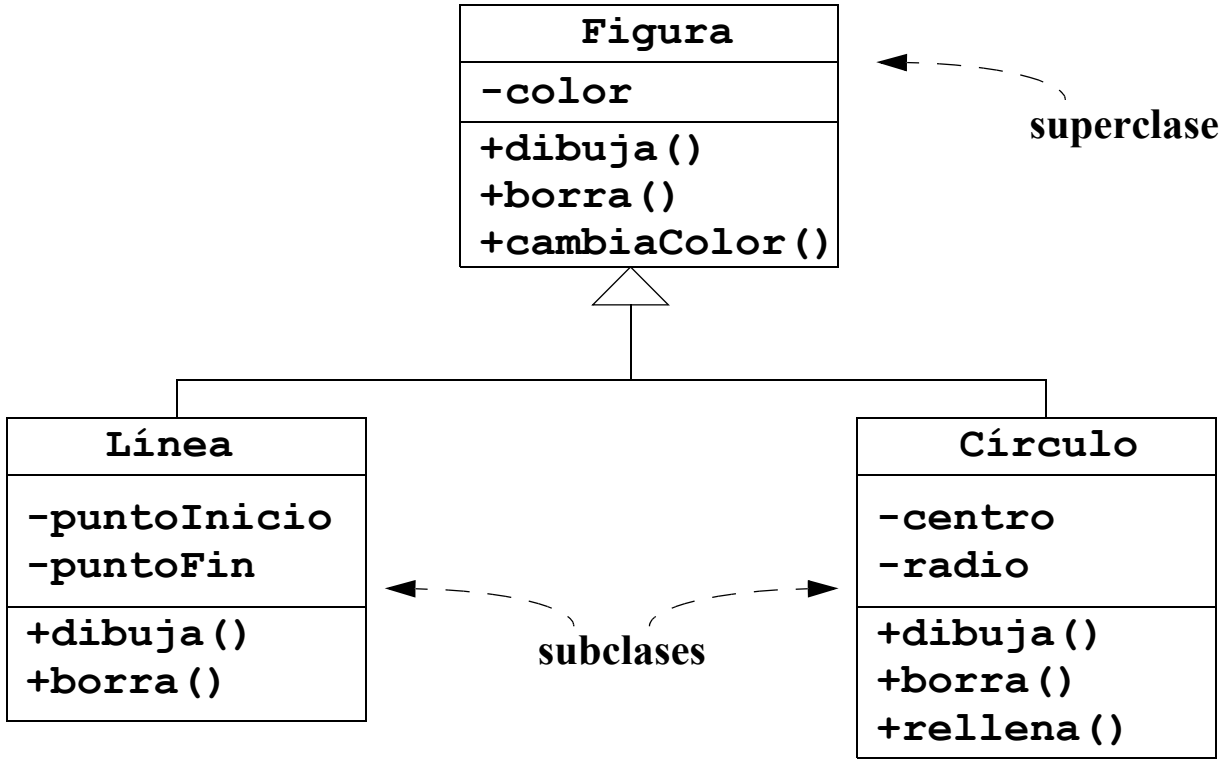
La nueva clase puede elegir para las operaciones *heredadas*:

- ***redefinir*** la operación: se vuelve a escribir
  - la nueva operación redefinida puede usar la del padre y hacer más cosas: programación *incremental*
  - o puede ser totalmente diferente
- dejarla como está: heredarla tal como está en el padre

La herencia se puede aplicar múltiples veces

- da lugar a una jerarquía de clases

Herencia en un diagrama de clases



- Los atributos y métodos de la superclase no se repiten en las subclases, salvo que se hayan redefinido
  - por ejemplo, `cambiaColor()` se hereda sin cambios
  - `rellena()` es una operación nueva

Herencia y Constructores

Los constructores no se heredan

- las subclases deben definir su propio constructor

Normalmente será necesario inicializar los atributos de la superclase

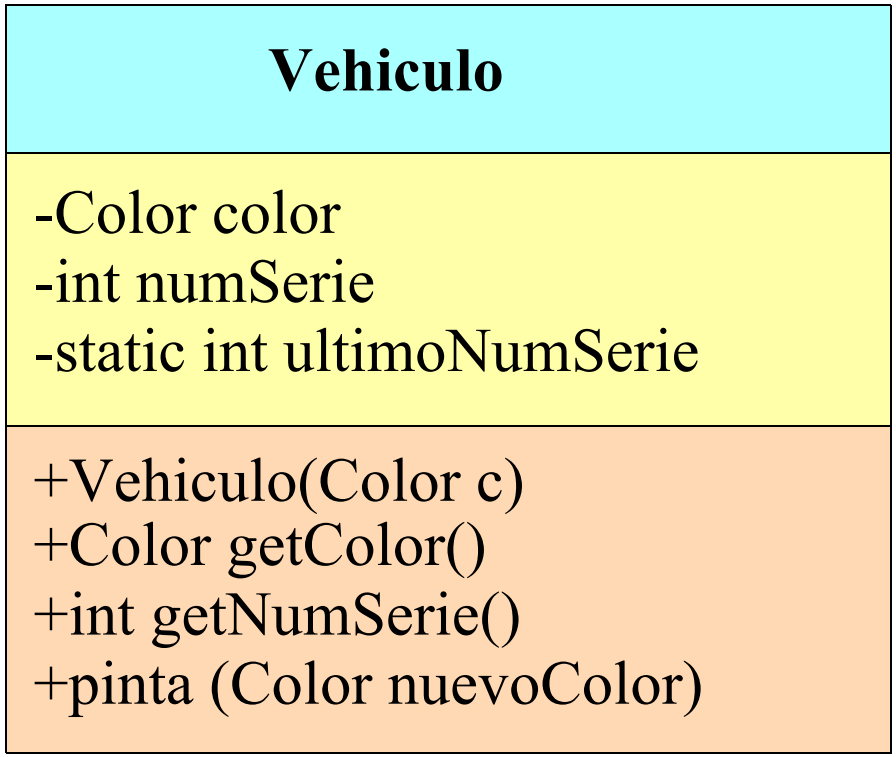
- para ello se llama al constructor de la superclase desde el de la subclase

```
/*
 * constructor de una subclase
 */
public Subclase(parámetros...) {
  // invoca el constructor de la superclase
  super(parámetros para la superclase);
  // inicializa sus propios atributos
  ...
}
```

- la llamada a "`super`" debe ser la primera instrucción del constructor de la subclase

Ejemplo

Clase que representa un vehículo cualquiera



Ejemplo: clase Vehiculo

```
/**
 * Clase que representa un vehículo cualquiera
 */
public class Vehiculo
{
    // colores de los que se puede pintar un vehículo
    // son de la clase enumerada Color
    public static enum Color {ROJO, VERDE, AZUL}

    // atributos
    private Color color;
    private final int numSerie;

    // atributo estático para obtener el núm. de serie
    private static int ultimoNumSerie=0;
```

clase Vehiculo (cont.)

```
/**
 * Construye un vehículo
 * @param c color del vehículo
 */
public Vehiculo(Color c)
{
  this.color=c;
  // obtener un nuevo número de serie
  ultimoNumSerie++;
  this.numSerie=ultimoNumSerie;
}
```

clase Vehiculo (cont.)

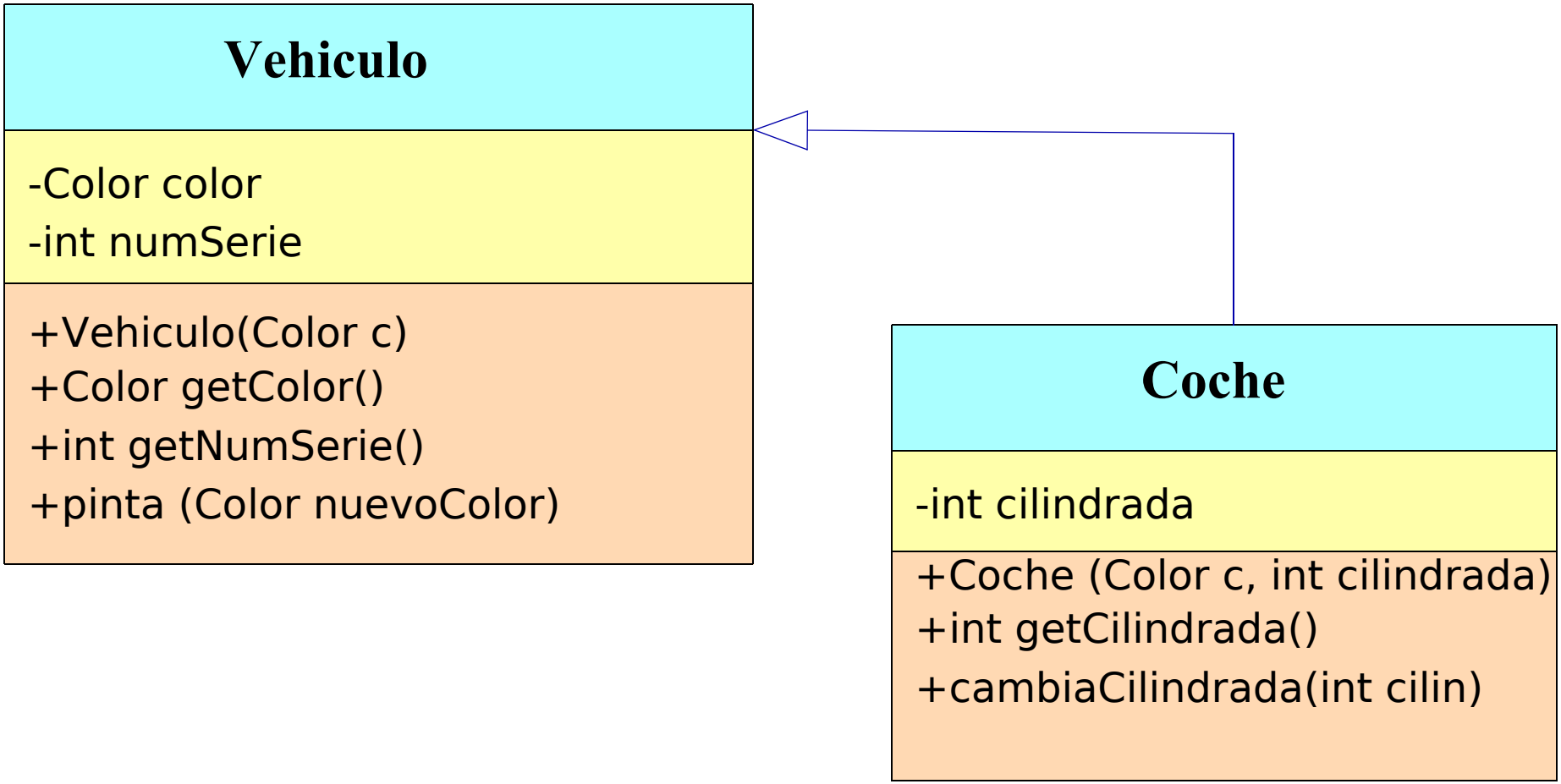
```
/**
 * Retorna el color del vehículo
 * @return color del vehículo
 */
public Color getColor()
{
  return color;
}

/**
 * Retorna el numero de serie del vehículo
 * @return numero de serie del vehículo
 */
public int getNumSerie()
{
  return numSerie;
}
```

clase Vehiculo (cont.)

```
   /**
    * Pinta el vehículo de un color
    * @param nuevoColor color con el que pintar
             el vehículo
    */
   public void pinta(Color nuevoColor)
   {
     color = nuevoColor;
   }
}
```

Ejemplo: extensión a la clase Coche



Extensión a la clase Coche (cont.)

```
/**
 * Clase que representa un coche
 */
public class Coche extends Vehiculo {
  // cilindrada del coche
  private int cilindrada;

  /**
   * Constructor al que le pasamos el color
   * y la cilindrada
   */
  public Coche(Color c, int cilindrada)
  {
    super(c);
    this.cilindrada = cilindrada;
  }
```

Extensión a la clase Coche (cont.)

```
  /** Obtiene la cilindrada del coche */
  public int getCilindrada(){
    return cilindrada;
  }

  /** Cambia la cilindrada del coche */
  public void cambiaCilindrada(int cilin) {
    this.cilindrada=cilin;
  }

  // Se heredan los métodos
  //   getColor(), getNumSerie() y pinta()
}
```

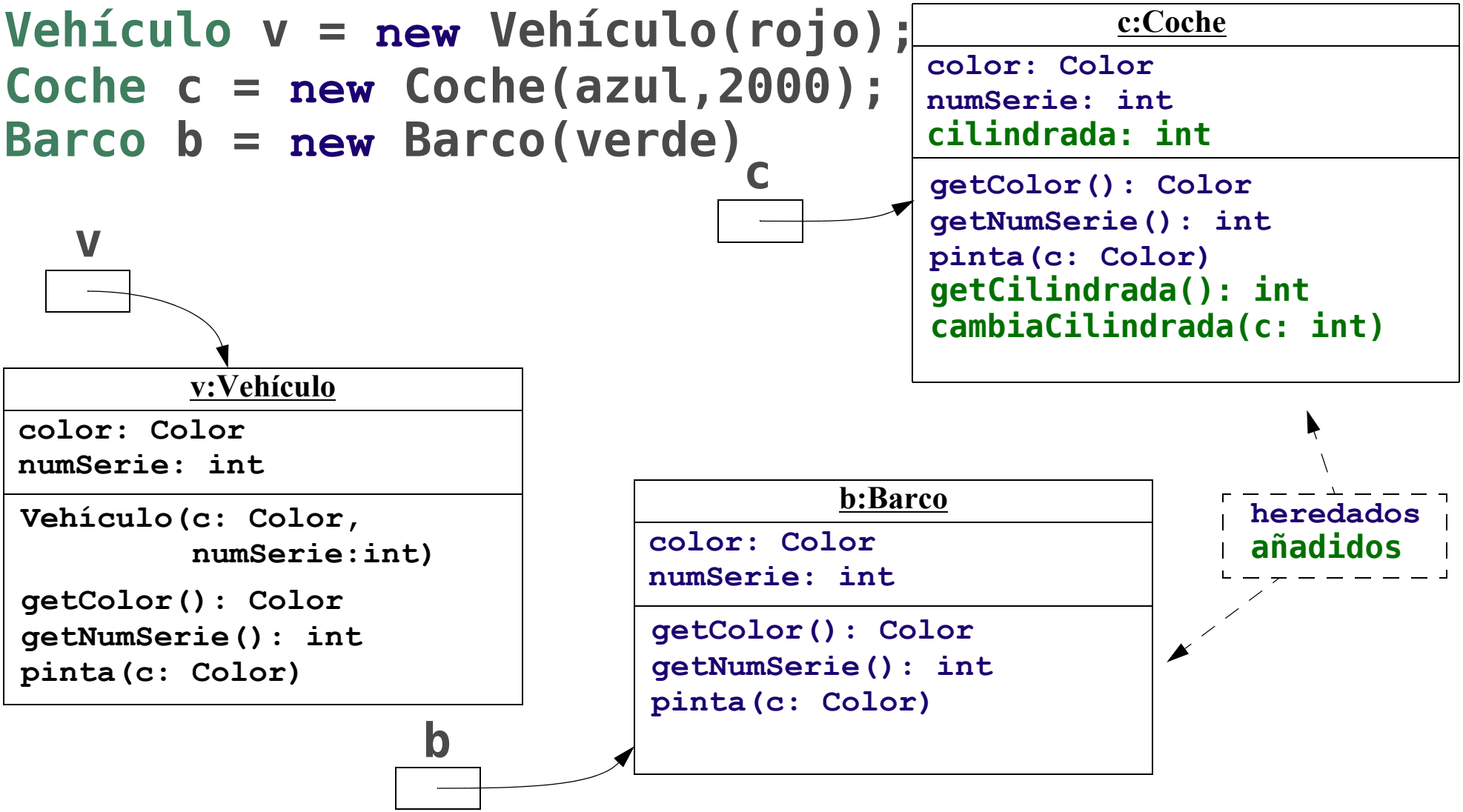
Ejemplo: extensión a la clase Barco

Se puede hacer herencia sin añadir métodos ni atributos

```
public class Barco extends Vehiculo
{
   /**
    * Constructor al que le pasamos el color
    */
   public Barco(Color c) {
     super(c);
   }
}
```

Ejemplo: objetos y herencia

```
Vehículo v = new Vehículo(rojo);
Coche c = new Coche(azul,2000);
Barco b = new Barco(verde)
```



Redefiniendo operaciones

Una subclase puede redefinir ("***override***") una operación en lugar de heredarla directamente

- es conveniente indicarlo poniendo `@override` justo antes del método
  - ello permitirá al compilador detectar errores en la cabecera del método

En muchas ocasiones (no siempre) la operación redefinida invoca la de la superclase

- se usa para ello la palabra `super`
- se refiere a la superclase directa del objeto actual

Invocación de un método de la superclase:

```
super.nombreMétodo(parametros...);
```

Ejemplo: nueva operación en la clase Vehiculo

```
public class Vehiculo {
    ...
    /**
     * Obtener un texto con los datos
     * del vehículo
     */
    public String toString()
    {
        return "Vehiculo -> numSerie= "+ numSerie+
               ", color= "+ color;
    }
}
```

Ejemplo: redefinir la nueva operación en la clase Coche

```
public class Coche {
   ...
   /**
    * Obtener un texto con los datos del coche,
    * incluyendo la cilindrada
    */
   @Override
   public String toString()
   {
       return super.toString()+", cilindrada= "+
              cilindrada;
   }
}
```

Resumen Herencia

Las clases se pueden ***extender***

- la subclase ***hereda*** los atributos y métodos de la superclase
  - aunque la parte privada de la superclase no es accesible desde las subclases (la parte `protected`, sí)

A la subclase se le pueden ***añadir*** nuevas operaciones y atributos

Al extender una clase se pueden ***redefinir*** sus operaciones

- si se desea, se puede invocar desde la nueva operación a la de la superclase: ***programación incremental***

Buena práctica de programación:

- utilizar `@Override` en los métodos redefinidos

Clases abstractas

En ocasiones definimos clases de las que no pretendemos crear objetos

- su único objetivo es que sirvan de superclases a las clases "reales"

Ejemplos:

- nunca crearemos objetos de la clase `Figura`
  - lo haremos de sus subclases `Círculo`, `Cuadrado`, ...
- nunca crearemos un `Vehiculo`
  - crearemos un `Coche`, un `Barco`, un `Avión`, ...

A ese tipo de clases las denominaremos ***clases abstractas***

Clases abstractas en Java

Las clases abstractas en Java se identifican mediante la palabra reservada `abstract`

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

Es un error tratar de crear un objeto de una clase abstracta

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

ERROR detectado por el compilador

Pero *NO* es un error utilizar referencias a clases abstractas

```
Figura f1 = new Círculo(...); // correcto
Figura f2 = new Cuadrado(...); // correcto
```

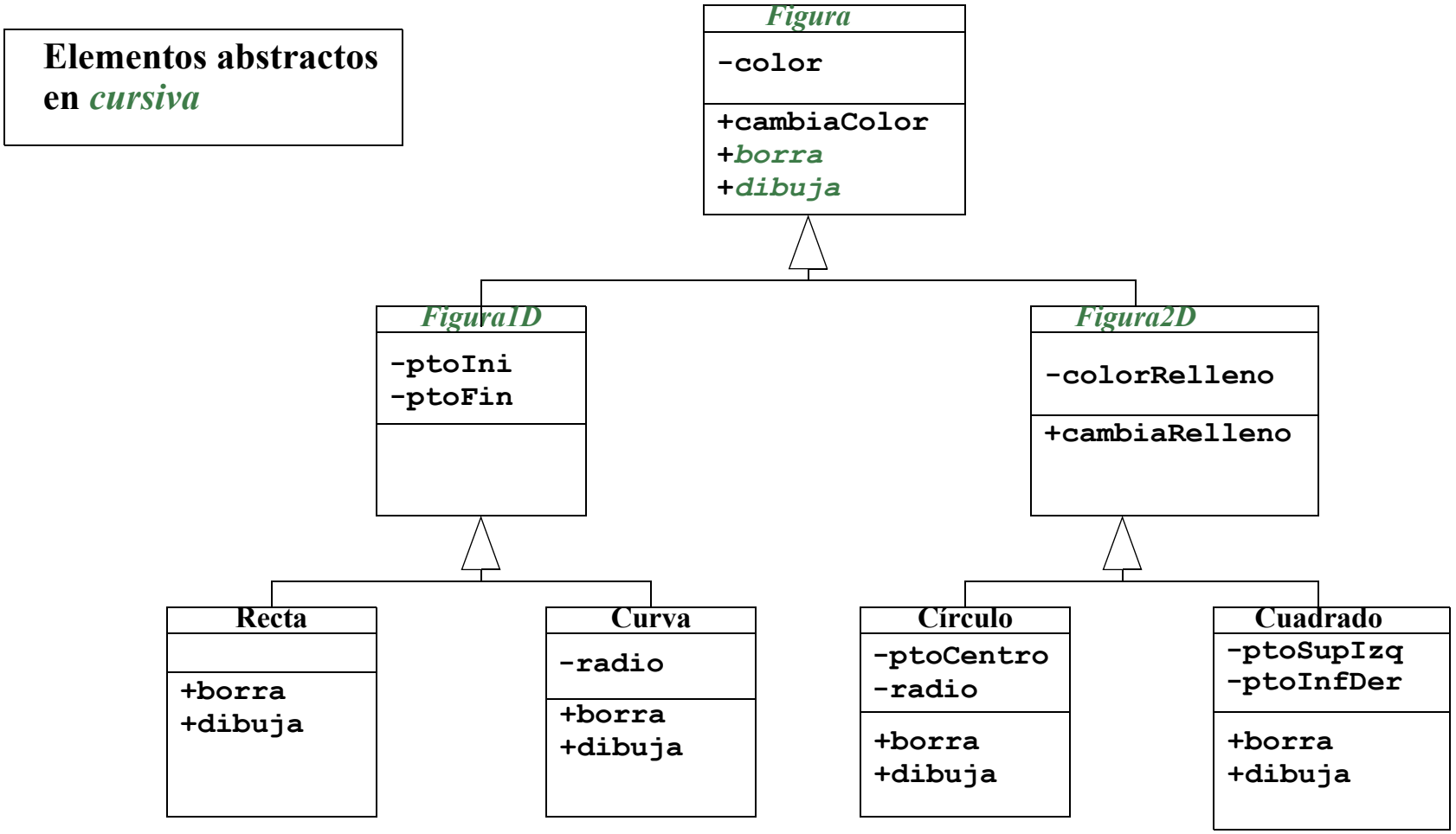
Métodos abstractos

Una clase abstracta puede tener métodos abstractos

- se trata de métodos sin cuerpo
- que ***es obligatorio redefinir*** en las subclases no abstractas
- ejemplo de método abstracto

```
public abstract void dibuja();
```

Ejemplo: clase abstracta `Figura` y subclases



Polimorfismo

Definición en *Wikipedia*: "proveer una interfaz común a entidades de tipos diferentes"

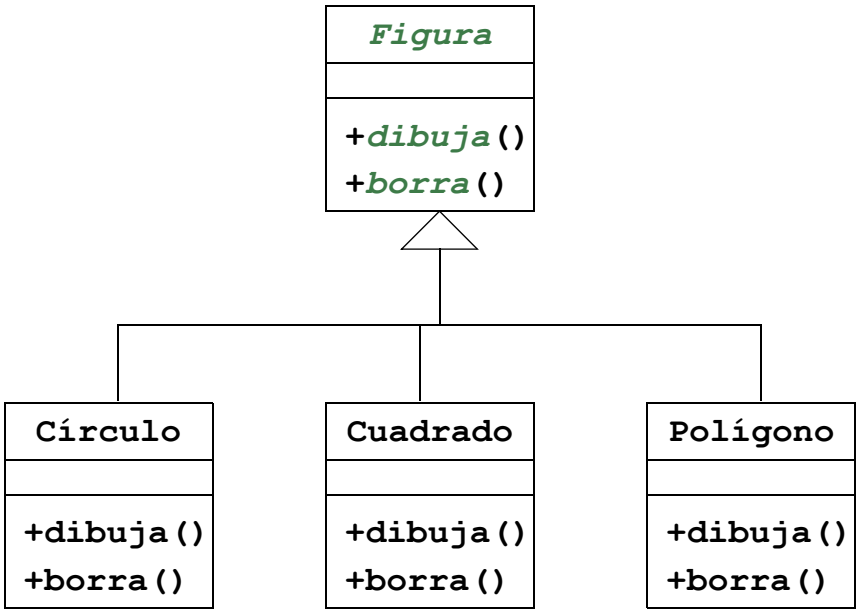
Las ***operaciones polimórficas*** son aquellas que, invocándose de la misma forma, hacen funciones similares con objetos de tipos diferentes

Ejemplo: suponer que existe la clase `Figura` y sus subclases

- `Círculo`
- `Cuadrado`
- `Polígono`

Todas ellas con las operaciones:

- `dibuja()`
- `borra()`



Polimorfismo (cont.)

Nos gustaría usar el ***polimorfismo*** para hacer la operación `mueveFigura` que opere correctamente con cualquier clase de figura:

```
mueveFigura(nuevaPosición):
  borra la figura
  dibuja la figura en la nueva posición
```

Esta operación debería:

- llamar a las operaciones `borra` y `dibuja` del `Círculo` cuando la figura sea un círculo
- llamar a las operaciones `borra` y `dibuja` del `Cuadrado` cuando la figura sea un cuadrado
- etc.

Si la programásemos con un `switch` sería largo si hay muchas figuras y *NO* funcionaría para figuras creadas en el futuro

Polimorfismo en Java

El polimorfismo en Java se basa en dos propiedades:

1. Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases

```
Vehiculo v1=new Coche(Vehiculo.rojo,12345,2000);
Vehiculo v2=new Barco(Vehiculo.azul,2345);
```

2. La operación se selecciona en base a la clase del objeto, no a la de la referencia

`v1.toString()` ← **usa el método de la clase `Coche`, puesto que `v1` es un coche**

`v2.toString()` ← **usa el método de la clase `Barco`, puesto que `v2` es un barco**

Polimorfismo en Java

Gracias a esas dos propiedades, el método `moverFigura` sería:

```
public void mueveFigura(Figura f, Posición pos){
  f.borra();
  f.dibuja(pos);
}
```

Y podría invocarse de la forma siguiente:

```
Círculo c = new Círculo(...);
Polígono p = new Polígono(...);
pos=...;

mueveFigura(c, pos);
mueveFigura(p, pos);
```

Polimorfismo en Java (cont.)

- Gracias a la primera propiedad el parámetro `f` puede referirse a cualquier subclase de `Figura`
- Gracias a la segunda propiedad en `mueveFigura` se llama a las operaciones `borra` y `dibuja` apropiadas

El lenguaje permite que una referencia a una superclase pueda apuntar a un objeto de cualquiera de sus subclases

- pero no al revés

```
Vehiculo v = new Coche(...); // permitido
Coche c = new Barco(...); // ¡NO permitido!
```

Ejemplo con array polimórfico

```
/**
 * Programa que tiene una lista de vehículos de
 * varias subclases. Muestra la lista en pantalla
 */
public class ListaVehiculos {
  public static void main(String[] args) {
    Vehiculo[] l=new Vehiculo[3];

    l[0]= new Coche(Vehiculo.Color.ROJO,1234,2000);
    l[1]= new Barco(Vehiculo.Color.VERDE, 222);
    l[2]= new Coche(Vehiculo.Color.AZUL,8234,3000);

    for (Vehiculo v: l) {
        System.out.println(v.toString());
    }
  }
}
```

lista polimórfica

llamada polimórfica

Conversión de referencias (*casting*)

Es posible convertir referencias

```
Vehiculo v=new Coche(...);
Coche c=(Coche)v; // casting

v.cilindrada(); // ¡ERROR!
c.cilindrada(); // correcto
```

El ***casting*** cambia el "*punto de vista*" con el que vemos al objeto

- a través de `v` le vemos como un `Vehiculo`
  - y por tanto sólo podemos invocar métodos de esa clase
- a través de `c` le vemos como un `Coche`
  - y podemos invocar cualquiera de los métodos de esa clase
- el objeto siempre es el mismo
  - tenemos un solo objeto coche, que a su vez es un solo vehículo

Resumen

El polimorfismo nos permite abstraer operaciones

- podemos invocarlas sin preocuparnos de las diferencias existentes para objetos diferentes
- el sistema elige la operación apropiada al objeto

El polimorfismo se asocia a las jerarquías de clases:

- una superclase y todas las subclases derivadas de ella

El polimorfismo en Java se basa en dos propiedades:

- Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases
- La operación se selecciona en base a la clase del objeto, no a la de la referencia