Prácticas de Programación

Tema 1. Introducción al análisis y diseño de programas

Tema 2. Clases y objetos

Tema 3. Herencia y Polimorfismo

Tema 4. Tratamiento de errores

Tema 5. Aspectos avanzados de los tipos de datos

Tema 6. Modularidad y abstracción: aspectos avanzados

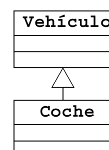
Tema 7. Entrada/salida con ficheros

Tema 8. Verificación y prueba de programas

Tema 3. Herencia y Polimorfismo

3.1. Herencia
3.2. Polimorfismo
3.3. La clase Object
3.4. Clases abstractas
3.5. Bibliografía

3.1 Herencia



Relación de herencia:

- *todos los* coches *son* vehículos

La herencia es un mecanismo por el que se pueden crear nuevas clases a partir de otras existentes,

- heredando, y posiblemente modificando, y/o añadiendo operaciones
- heredando y posiblemente añadiendo atributos

Observar que una operación o atributo no puede ser suprimido en el mecanismo de herencia

Nomenclatura

clase original	superclase	padre	Vehículo
clase extendida	subclase	hijo	Coche

Herencia de operaciones

Al extender una clase

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

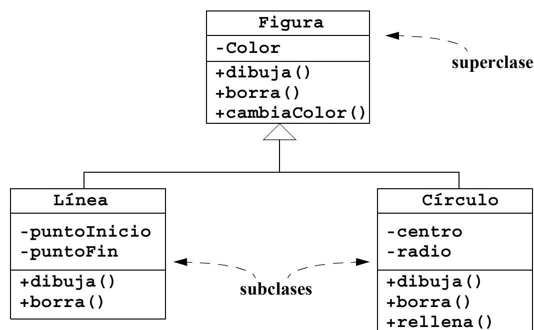
La nueva clase puede elegir para las operaciones heredadas:

- *redefinir* la operación: se vuelve a escribir
  - la nueva operación puede usar la del padre y hacer más cosas: programación incremental
  - o puede ser totalmente diferente
- dejarla como está: no hacer nada

La herencia se puede aplicar múltiples veces

- da lugar a una jerarquía de clases

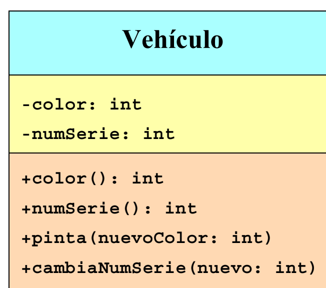
Herencia en un diagrama de clases



- Los atributos y métodos de la superclase no se repiten en las subclases
  - salvo que se hayan redefinido

Ejemplo

Clase que representa un vehículo cualquiera



Ejemplo: clase Vehículo

```
public class Vehículo
{
  // constantes estáticas finales para los colores
  public static final int
    rojo=1,
    verde=2,
    azul=3;

  // atributos
  private int color;
  private int numSerie;
```

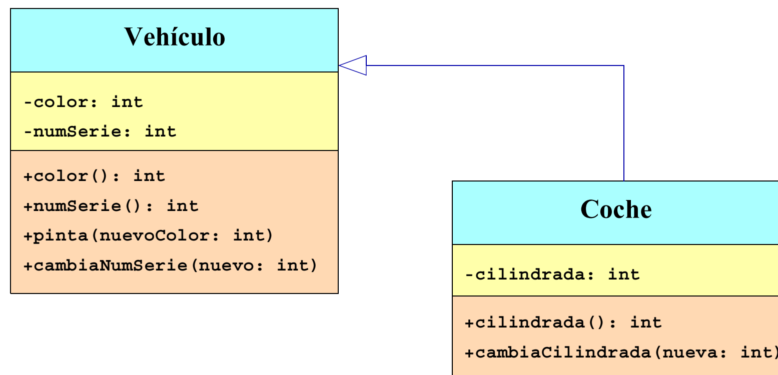


```
  /**
   * Obtener el numero de serie
   */
  public int numSerie()
  {
    return numSerie;
  }


  /**
   * Obtener el color
   */
  public int color()
  {
    return color;
  }
```



```
  /**
   * Cambiar el numero de serie
   */
  public void cambiaNumSerie(int numSerie)
  {
    this.numSerie=numSerie;
  }


  /**
   * Pintar el vehículo de un color
   */
  public void pinta(int nuevoColor)
  {
    color = nuevoColor;
  }
}
```

Ejemplo: extensión a la clase Coche



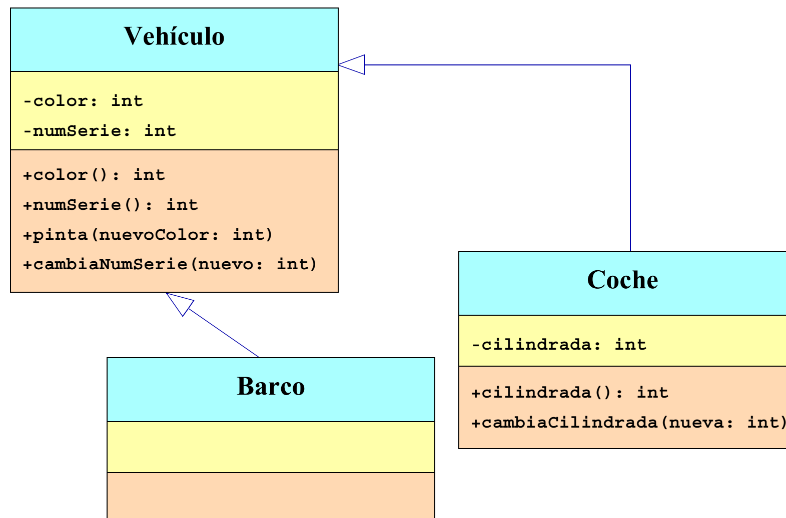

```
public class Coche extends Vehículo
{
  // cilindrada del coche
  private int cilindrada;

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

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


Ejemplo: extensión a la clase Barco

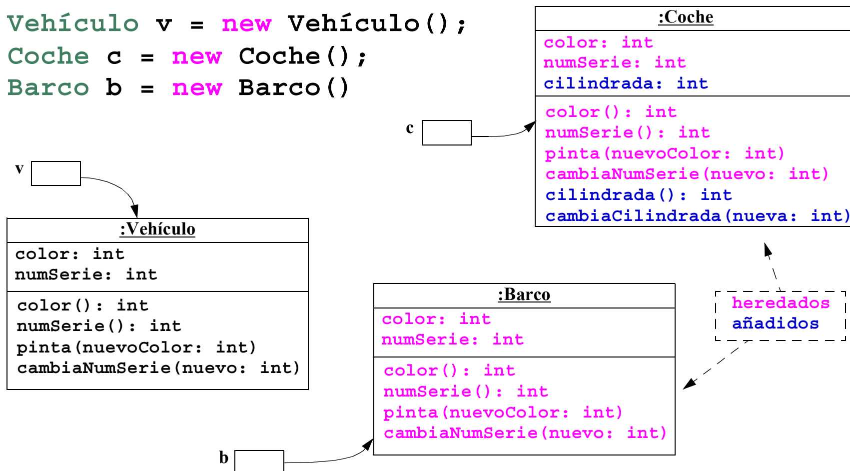
Se puede hacer herencia sin añadir métodos y/o atributos

```
public class Barco extends Vehículo
{

}
```

Ejemplo: jerarquía de clases



Ejemplo: objetos y herencia



Redefiniendo operaciones

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

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 del objeto actual

Invocación del constructor de la superclase:

```
super(parámetros...);
```

Invocación de un método de la superclase:

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

Herencia y Constructores

Los constructores no se heredan

- **las subclases deben definir su propio constructor**

Normalmente será necesario inicializar los atributos de la superclase

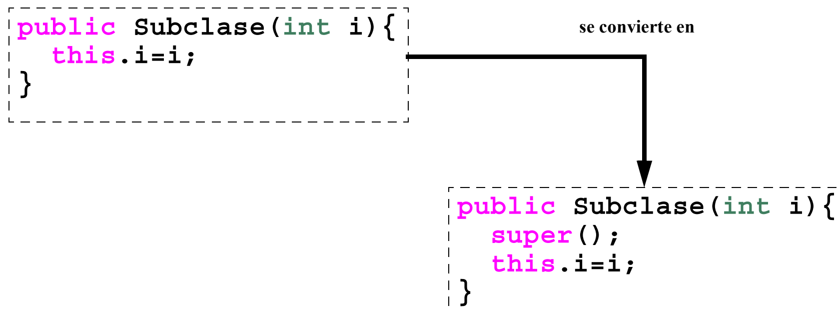
- **para ello se llama a su constructor desde el de la subclase**

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

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

Si desde un constructor de una subclase no se llama expresamente al de la superclase

- **el compilador añade la llamada al constructor sin parámetros**



```
public Subclase(int i){
   this.i=i;
}
```

se convierte en

```
public Subclase(int i){
   super();
   this.i=i;
}
```

- **en el caso de que la superclase no tenga un constructor sin parámetros se produciría un error de compilación**

Ejemplo: constructores y redefinición de operaciones

Modificamos la clase `Vehículo` para añadir un constructor y un método que retorna en un `String` los datos del objeto

Vehículo
-color: int -numSerie: int
+Vehiculo(color: int, numSerie: int) +numSerie(): int +color(): int +pinta(nuevoColor: int) +toString(): String

(En el capítulo 3.3 veremos porqué llamamos `toString` al método)

Ejemplo: clase Vehículo

```
public class Vehículo {
    // constantes estáticas para los colores
    public static final int rojo=1, verde=2, azul=3;
    public static final String[] nombre=
        {"error", "rojo", "verde", "azul"};
    // atributos privados
    private int color;
    private int numSerie;

    /**
     * Constructor al que le pasamos el color
     * y el número de serie
     */
    public Vehículo(int color, int numSerie) {
        this.color = color;
        this.numSerie = numSerie;
    }
```



```
    public int numSerie() {...}

    public int color() {...}

    public void cambiaNumSerie(int numSerie)  {...}

    public void pinta(int nuevoColor)   {...}

    /**
     * Obtener un texto con los datos
     * del vehículo
     */
    public String toString() {
        return "Vehículo -> numSerie= "+
               numSerie+", color= "+nombre[color];
    }

}
```

No repetimos el código (es igual que en el ejemplo anterior)

Ejemplo: Extensión al Coche

```
public class Coche extends Vehículo {

    // cilindrada del coche
    private int cilindrada;

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

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

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

   /** Retorna un texto con los datos del coche */
   @Override
   public String toString() {
        return super.toString()+
           ", cilindrada= " + cilindrada;
   }

}
```


Ejemplo: extensión al Barco

```
public class Barco extends Vehículo {

   /**
    * Constructor al que le pasamos el color y
    * el numero de serie
    */
   public Barco(int color, int numSerie) {
     super(color, numSerie);
   }

}
```


Modificador de acceso `protected`

Modificadores de acceso para miembros de clases:

- **<*ninguno*>: accesible desde el paquete**
- `public`**: accesible desde todo el programa**
- `private`**: accesible sólo desde esa clase**
- *`protected`***: accesible desde el paquete y desde sus subclases en cualquier paquete**

Definir atributos `protected` NO es una buena práctica de programación

- **ese campo sería accesible desde cualquier subclase**
  - **puede haber muchas y eso complicaría enormemente la tarea de mantenimiento**
- **además el campo es accesible desde todas las clases del paquete (subclases o no)**

Uso recomendado del modificador de acceso `protected`

- regla general: todos los campos de una clase son privados
- se proporcionan métodos públicos para leer y/o cambiar los campos (pero sólo cuando sea necesario)
- en el caso de que se desee que un campo *sólo* pueda ser leído y/o cambiado por las subclases se hacen métodos `protected`

```
public class Superclase {
  private int atributo; // atributo privado
  // método para leer (público)
  public int atributo() {
    return atributo;
  }
  // método para cambiar (sólo para las subclases)
  protected void cambiaAtributo(int a) {
    atributo = a;
  }
```


Resumen Herencia

Las clases se pueden *extender*

- la subclase *hereda* los atributos y métodos de la superclase

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*

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

Buena práctica de programación:

- utilizar `@Override` en los métodos redefinidos

3.2 Polimorfismo

La palabra polimorfismo viene de “múltiples formas”

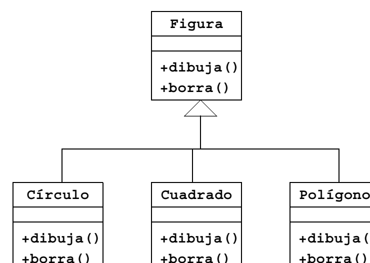
Las *operaciones polimórficas* son aquellas que hacen funciones similares con objetos diferentes

Ejemplo:

- suponer que existe la clase `Figura` y sus subclases
  - Círculo
  - Cuadrado
  - Polígono

Todas ellas con las operaciones:

  - dibuja
  - borra



Nos gustaría poder hacer la *operación polimórfica* `mueveFigura` que opere correctamente con cualquier clase de figura:

```
mueveFigura
  borra
  dibuja en la nueva posición
```

Esta operación polimórfica debería:

- llamar a la operación `borra` del `Círculo` cuando la figura sea un círculo
- llamar a la operación `borra` del `Cuadrado` cuando la figura sea un cuadrado
- etc.

Polimorfismo en Java

El polimorfismo en Java consiste en dos propiedades:

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

```
Vehículo v1=new Coche(Vehiculo.rojo,12345,2000);
Vehículo 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
```


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(...);

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

- 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

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

Justificación:

- un coche es un vehículo
  - cualquier operación de la clase `Vehículo` existe (sobrescrita o no) en la clase `Coche`
    ```
    v.cualquierOperación(...); // siempre correcto
    ```
- un vehículo no es un coche
  - sería un error tratar de invocar la operación:
    ```
    c.cilindrada(); // ERROR: cilindrada() no
                    // existe para un vehículo
    ```
  - por esa razón el lenguaje lo prohíbe

Conversión de referencias (*casting*)

Es posible convertir referencias

```
Vehículo v=new Coche(...);
Coche c=(Coche)v;

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 `Vehículo` (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)

Hacer una conversión de tipos incorrecta produce una excepción `ClassCastException` en tiempo de ejecución

```
Vehículo v=new Vehículo(...);
Coche c=(Coche)v;
```

lanza `ClassCastException` en tiempo de ejecución

Java proporciona el operador `instanceof` que permite conocer la clase de un objeto

```
if (v instanceof Coche) {
  Coche c=(Coche)v;
  ...
}
```

NUNCA lanza `ClassCastException` (por que es seguro que `v` es un coche)

- “`v instanceof Coche`” retorna `true` si `v` apunta a un objeto de la clase `Coche` o de cualquiera de sus (posibles) subclases

Arrays de objetos de distintos tipos

Gracias al polimorfismo es posible que un array contenga referencias a objetos de distintas clases

- **la superclase y todas sus subclases**

Ejemplo: array de figuras

```
Figura[] figuras = new Figura[3];
figuras[0] = new Círculo(...);
figuras[1] = new Cuadrado(...);
figuras[2] = new Polígono(...);

// borra todas las figuras
for(int i=0; i<figuras.length; i++)
  figuras[i].borra();
```

Llama a la operación `borra` correspondiente a la clase del objeto

Ejemplo: registro de vehículos

```
public class RegistroDeVehículos {
  // almacena todos los vehículos registrados
  private Vehículo[] vRegistrados;
  // número de vehículos registrados
  private int numVehículos;
  /** Constructor */
  public RegistroDeVehículos(int maxNumVehículos) {
    vRegistrados=new Vehículo[maxNumVehículos];
    numVehículos=0;
  }
  /** registra un vehículo */
  public boolean registraVehículo(Vehículo v) {
    if (numVehículos == vRegistrados.length)
      return false;
    vRegistrados[numVehículos++]=v;
    return true;
  }
```



```
  /** elimina del registro el vehículo con el
   * número de serie indicado
   * Retorna false si no le encuentra */
  public boolean eliminaVehículo(int numSerie) {
    for(int i=0; i<numVehículos; i++)
      if (numSerie==vRegistrados[i].numSerie()) {
        // Encontrado. Se elimina desplazando los
        // siguientes "hacia la izquierda"
        for(int j=i; j<numVehículos-1; j++)
          vRegistrados[j]=vRegistrados[j+1];
        numVehículos--;
        return true;
      } // fin if

    // si llega aquí es porque no le ha encontrado
    return false;
  }
```

```java
/** retorna el vehículo que ocupa la posición
 * "pos" del registro */
public Vehículo vehículoEnPos(int pos) {
  if (pos<0 || pos>=numVehículos)
    return null; // posición inválida
  // posición válida, retorna el vehículo
  return vRegistrados[pos];
}


/** pinta todos los vehículos */
public void pintaTodos(int color) {
  for(int i=0; i<numVehículos; i++)
    vRegistrados[i].pinta(color);
}
```



```java
  /** retorna string con todos los vehículos */
  public String toString() {
    String txt="";
    for(int i=0; i<numVehículos; i++)
      txt = txt + vRegistrados[i].toString() + "   ";
    return txt;
  }

} // clase RegistroDeVehículos
```


Ejemplo de utilización de la clase `RegistroDeVehículos`:

```java
RegistroDeVehículos registro =
                      new RegistroDeVehículos(5);
Barco b = new Barco(Vehículo.verde, 1274);
registro.registraVehículo(b);
registro.registraVehículo(new Coche(Vehículo.azul,
                                    3021, 2000));
registro.registraVehículo(new Barco(Vehículo.rojo,
                                    4765));
registro.pintaTodos(Vehículo.verde);
System.out.println(registro.todosATexto());
Vehículo v = registro.vehículoEnPos(1);
```

Si creamos la clase `Avión` que extiende a `Vehículo`

- **la clase `RegistroDeVehículos` *no necesita ser modificada***

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:

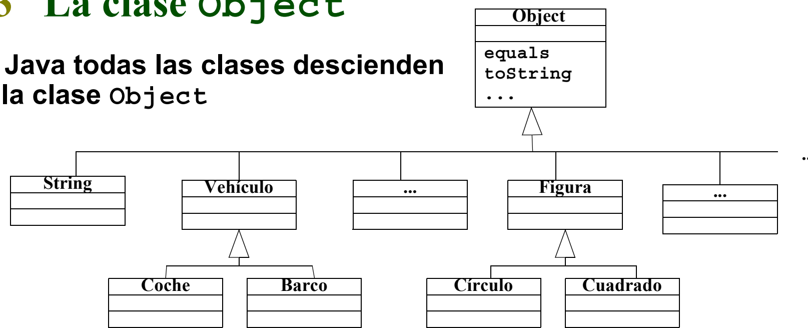
- una superclase y todas las subclases derivadas de ella directa o indirectamente

El polimorfismo en Java consiste en dos propiedades:

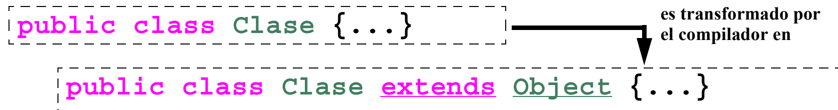
1. Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases
2. La operación se selecciona en base a la clase del objeto, no a la de la referencia

3.3 La clase `Object`

En Java todas las clases descienden de la clase `Object`



Es como si el compilador añadiera "`extends Object`" a todas las clases de primer nivel



Método `equals`

Se encuentra definido en la clase `Object` como:

```
public class Object {
   ...
   public boolean equals(Object obj) {
     return this == obj;
   }
   ...
}
```

- es decir, compara referencias, no contenidos

Como cualquier otro método de una superclase

- *se puede redefinir en sus subclases*

Con lo que sabemos ahora ya podemos entender la redefinición del método `equals` para la clase `Coordenada` (vista en el tema 2):

```
public class Coordenada {
  private int x; // coordenada en el eje x
  private int y; // coordenada en el eje y

  ...

  @Override
  public boolean equals(Object obj) {
    Coordenada c = (Coordenada) obj;
    return c.x == x && c.y == y;
  }
}
```

- `@Override`: aconsejable cuando se redefine un método para detectar errores
- `public boolean equals(Object obj)`: Redefine el método `equals` de la clase `Object`
- `Coordenada c = (Coordenada) obj;`: Cambio de "punto de vista" para poder acceder a los campos `x` e `y` de `obj`

Para ser más correctos, la redefinición del método debería ser:

```
public boolean equals(Object obj) {
  if (!(obj instanceof Coordenada))
    return false;
  Coordenada c = (Coordenada) obj;
  return c.x == x && c.y == y;
}
```

- `if (!(obj instanceof Coordenada))`: Si `obj` no es de la clase `Coordenada` retorna `false` directamente y evita la excepción `ClassCastException`

Muchas clases estándar Java utilizan el método `equals` de la clase `Object` para comparar objetos

- por esa razón es importante que nuestras clases redefinan este método en lugar de definir uno similar

```
public boolean equals(Coordenada obj){
  ...
}
```

- `public boolean equals(Coordenada obj)`: NO redefine el método `equals` de la clase `Object`

Método `toString`

Se encuentra definido en la clase `Object` como:

```
public class Object {
  ...
  public String toString() {
    return ...;
  }
  ...
}
```

- es utilizado por el sistema cuando se concatena un objeto con un string, por ejemplo:

```
println("Valor coordenada:" + c);
```

- por defecto retorna un string con el nombre de la clase y la dirección de memoria que ocupa el objeto

```
Coordenada@a34f5bd
```

Una redefinición útil del método `toString` para la clase `Coordenada` podría ser:

```
@Override
public String toString() {
   return "(" + x + "," + y + ")";
}
```

Con esta redefinición el segmento de código

```
Coordenada c = new Coordenada(1,2);
System.out.println("Coord: " + c);
```

- produciría la salida:

```
Coord:(1,2)
```


3.4 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 `Vehículo`
  - crearemos un `Coche`, un `Barco`, un `Avión`, ...

La razón es que no existen "figuras" o "vehículos" genéricos

- ambos conceptos son abstracciones de los objetos reales, tales como círculos, cuadrados, coches o aviones
- 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

- que pueden apuntar a objetos de cualquiera de sus subclases (como ocurría con las referencias a superclases no abstractas)

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

Métodos abstractos

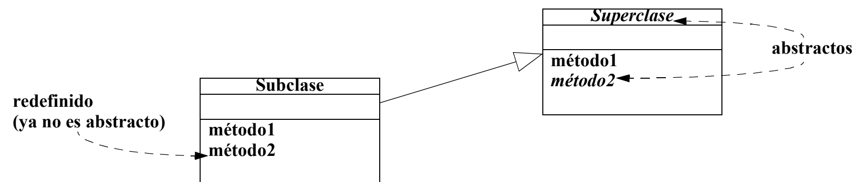
Una clase abstracta puede tener métodos abstractos

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

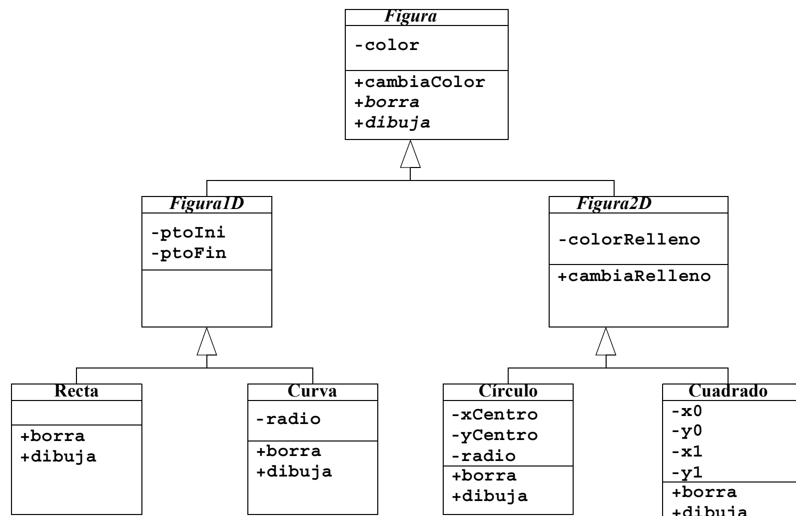
```
public abstract int métodoAbstracto(double d);
```

no tiene cuerpo

En el diagrama de clases, las clases y los métodos abstractos se escriben en cursiva



Ejemplo: clase abstracta `Figura` y subclases




```
public abstract class Figura {
    // color del borde de la figura
    private int color;
    /** Constructor */
    public Figura(int color) {
        this.color=color;
    }
    /** cambia el color del borde de la figura */
    public void cambiaColor(int color) {
        this.color=color;
    }
    /** borra la figura (abstracto) */
    public abstract void borra();
    /** dibuja la figura (abstracto) */
    public abstract void dibuja();
}
```

```java
public abstract class Figura1D extends Figura {

  /** Constructor */
  public Figura1D(int color){
    super(color);
  }

  // NO redefine ningún método abstracto

}
```



```java
public abstract class Figura2D extends Figura {

  // color de relleno de la figura
  private int colorRelleno;

  /** Constructor */
  public Figura2D(int color, int colorRelleno) {
    super(color);
    this.colorRelleno=colorRelleno;
  }

  /** cambia el color de relleno */
  public void cambiaRelleno(int color) {
    colorRelleno=color;
  }

  // NO redefine ningún método abstracto
}
```



```java
public class Recta extends Figura1D {
  private final double x0,y0,x1,y1;
  /** Constructor */
  public Recta(int color, double x0, double y0,
               double x1, double y1) {
    super(color);
    this.x0=x0; this.y0=y0; this.x1=x1; this.y1=y1;
  }
  /** implementa el método abstracto borra */
  @Override
  public void borra() { implementación...; }

  /** implementa el método abstracto dibuja */
  @Override
  public void dibuja() { implementación...; }
    ...;
  }
}
```

Ejemplo: clase abstracta Figura y subclases (cont.)

```
public class Círculo extends Figura2D {
  private double xCentro, yCentro, radio;
  /** Constructor */
  public Círculo(int color, int colorRelleno,
        double xCentro,double yCentro,double radio){
    super(color, colorRelleno);
    this.xCentro=xCentro; this.yCentro=yCentro;
    this.radio=radio;
  }
  /** implementa el método abstracto borra */
  @Override
  public void borra() {
    implementación...;
  }
  /** implementa el método abstracto dibuja */
  @Override
  public void dibuja() { implementación...; }
}
```


3.5 Bibliografía

- **King, Kim N. “Java programming: from the beginning”. W. W. Norton & Company, cop. 2000**
- **Francisco Gutiérrez, Francisco Durán, Ernesto Pimentel. “Programación Orientada a Objetos con Java”. Paraninfo, 2007.**
- **Ken Arnold, James Gosling, David Holmes, “El lenguaje de programación Java”, 3ª edición. Addison-Wesley, 2000.**
- **Eitel, Harvey M. y Deitel, Paul J., “Cómo programar en Java”, quinta edición. Pearson Educación, México, 2004.**