# Métodos de Programación







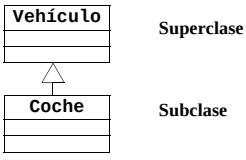
# Tema 5. Herencia y Polimorfismo







## 5.1 Herencia




Superclase

Subclase

**Relación de herencia:**

- ***todos los*** **coches** ***son*** **vehículos**

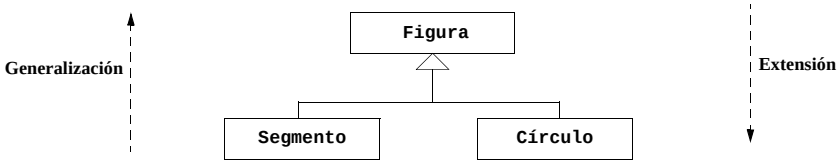
**La herencia es un mecanismo que permite crear nuevas clases (subclases) a partir de otras existentes (superclases)**

**Una subclase:**

- ***Hereda*** **todos los atributos y operaciones de la superclase**
- ***Puede añadir*** **nuevos atributos y operaciones**
- ***Puede redefinir*** **algunas operaciones para que tengan un comportamiento diferente de las heredadas**

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

## La *Herencia* también se denomina *Extensión* o *Generalización*




### Nomenclatura

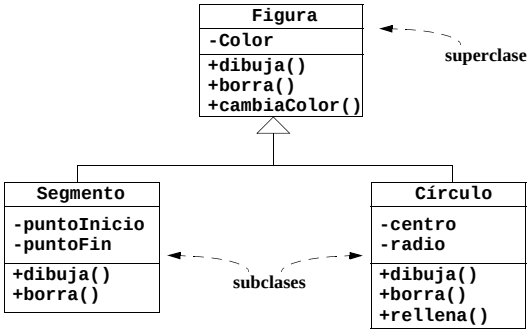
| | | | |
|---|---|---|---|
| **superclase** | **clase original** | **padre** | `Figura` |
| **subclase** | **clase extendida** | **hijo** | `Segmento, Círculo` |

## La *Herencia* (y el Polimorfismo) son unos de los *conceptos más importantes y diferenciadores* de la Programación Orientada a Objetos





# Herencia en un diagrama de clases




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

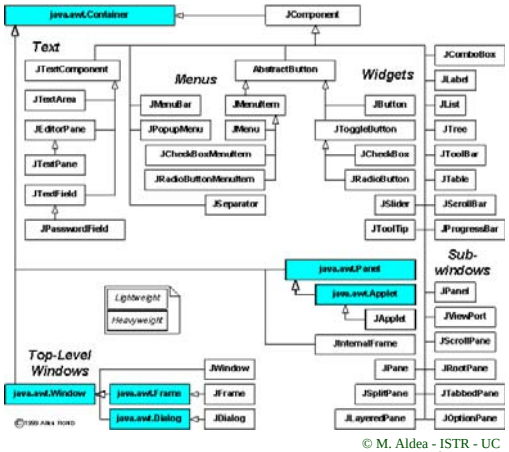




# Jerarquías de clases

## La herencia puede aplicarse en sucesivos niveles

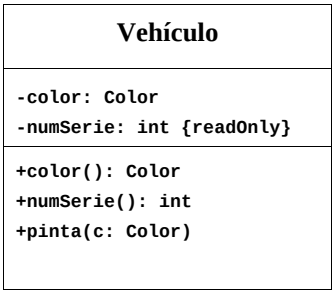
- **creando grandes *jerarquías de clases***




**Jerarquía de clases de la biblioteca gráfica Swing**

# Ejemplo sencillo

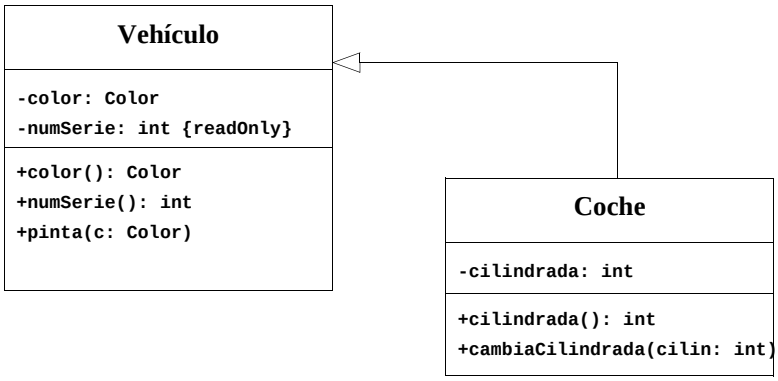
**Clase que representa un vehículo cualquiera**








# Ejemplo sencillo: subclase `Coche`

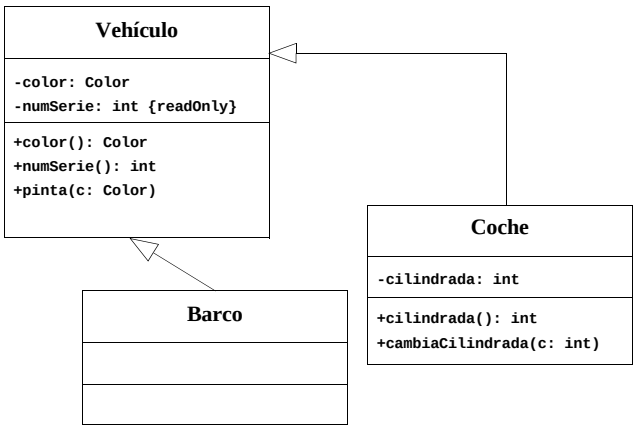



- **La clase `Coche` añade el atributo `cilindrada` y los métodos para gestionar dicho atributo**





# Ejemplo sencillo: subclase `Barco`




**(Observar que se puede extender una clase sin añadir atributos ni métodos)**

# Implementación del ejemplo en Java

```
/**
 * Clase que representa un vehículo cualquiera.
 */
public class Vehículo {
   // colores de los que se puede pintar un vehículo
   public static enum Color {ROJO, VERDE, AZUL}

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

   /**
    * Construye un vehículo.
    * @param color color del vehículo
    * @param numSerie número de serie del vehículo
    */
   public Vehículo(Color color, int numSerie) {
      this.color = color;
      this.numSerie = numSerie;
   }
```





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

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

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





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

  public Coche(...) {
    hablaremos del constructor más adelante
  }

  /** Retorna la cilindrada del coche. … */
  public int cilindrada() {
    return cilindrada;
  }

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

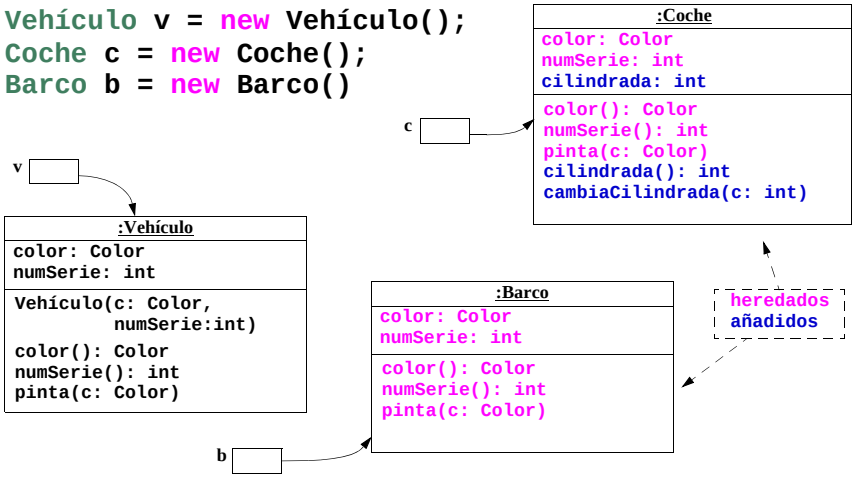
```
public class Barco extends Vehículo {

  public Barco(...) {
    hablaremos del constructor más adelante
  }

}
```





# Ejemplo: objetos y herencia

```
Vehículo v = new Vehículo();
Coche c = new Coche();
Barco b = new Barco()
```








# Herencia y Constructores en Java

**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**

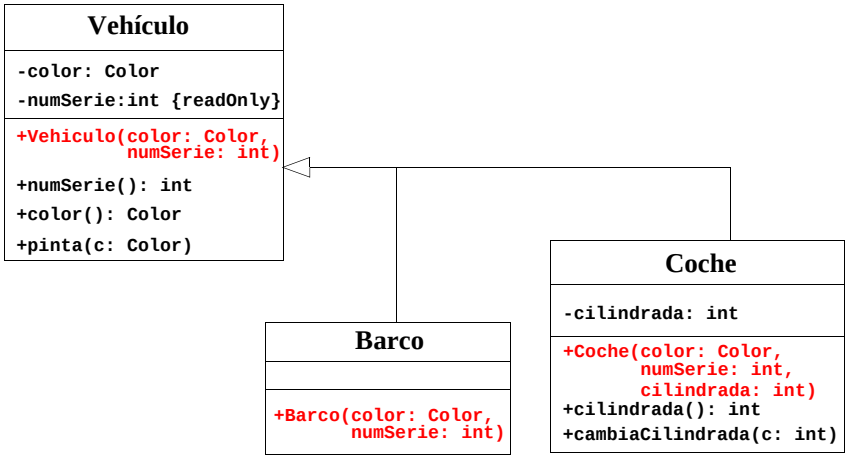






# Ejemplo: Constructores en Java

**Añadimos los constructores a la jerarquía de vehículos**










```
public class Vehículo {
   // colores de los que se puede pintar un vehículo
   public static enum Color {ROJO, VERDE, AZUL}
   // atributos privados
   private Color color;
   private final int numSerie;

  /**
   * Construye un vehículo.
   * @param color color del vehículo
   * @param numSerie número de serie del vehículo
   */
   public Vehículo(Color color, int numSerie) {
       this.color = color;
       this.numSerie = numSerie;
   }

   ... otros métodos ...
}
```

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

  /**
   * Construye un coche.
   * @param color color del coche
   * @param numSerie número de serie del coche
   * @param cilindrada cilindrada del coche
   */
  public Coche(Color color, int numSerie,
               int cilindrada) {
    super(color, numSerie);
    this.cilindrada = cilindrada;
  }

   ... otros métodos ...
}
```





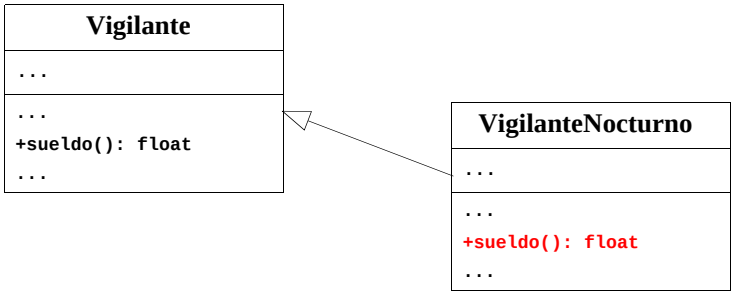
```
public class Barco extends Vehículo {

  /**
   * Construye un barco.
   * @param color color del barco
   * @param numSerie número de serie del barco
   */
  public Barco(Color color, int numSerie) {
    super(color, numSerie);
  }

}
```





# Redefiniendo operaciones

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




**La operación redefinida:**

- **puede ser totalmente diferente de la heredada**
- **o usar la de la superclase y hacer más cosas**

# Redefiniendo métodos en Java

**Es conveniente indicar que se desea redefinir un método utilizando la anotación `@Override`**

- **para informar al compilador de que el método redefine uno de la superclase**
- **y detectar el error en caso de que no lo haga**

**Ejemplo: redefinición errónea del método `sueldo()`**

```
public class VigilanteNocturno extends Vigilante {
  ...

  @Override
  public float suelo() {
    ...
  }
}
```

**Gracias a la anotación `@Override`, el compilador nos informa de que `suelo()` NO redefine ningún método de `Vigilante`**

**Por error hemos escrito `suelo()` en lugar de `sueldo()`**







# Invocando operaciones de la superclase

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

**Ejemplo:**

- **El sueldo de un `VigilanteNocturno` es el sueldo de un `Vigilante` más un plus por nocturnidad.**

**El lenguaje de programación debe proporcionarnos algún mecanismo para invocar las operaciones de la superclase**







# Invocando métodos de la superclase en Java

**Para referirse a un método de la superclase del objeto actual**

- **se usa la palabra reservada `super`**

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

**Ejemplo:**

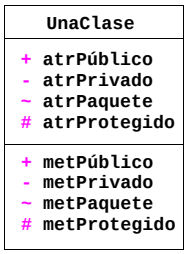
```
public class VigilanteNocturno extends Vigilante {
  ...

  @Override
  public float sueldo() {
    return super.sueldo() + PLUS_NOCTURNIDAD;
  }
}
```

**Sueldo de un Vigilante**

# 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 sus subclases y, en Java, desde cualquier clase en el mismo paquete**



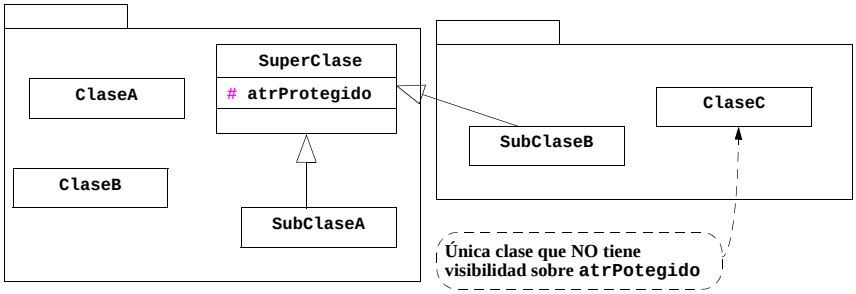







**En general, definir *atributos protected en Java NO es una buena práctica* de programación**

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










**Uso recomendado del modificador de acceso `protected`**

- **encapsular todas las clases de la jerarquía en el mismo paquete**
- **para los atributos que *sólo* puedan ser leídos y/o cambiados 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;
  }
}
```

- ***No se debe abusar* del modificador de acceso `protected`**

# 5.2 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**

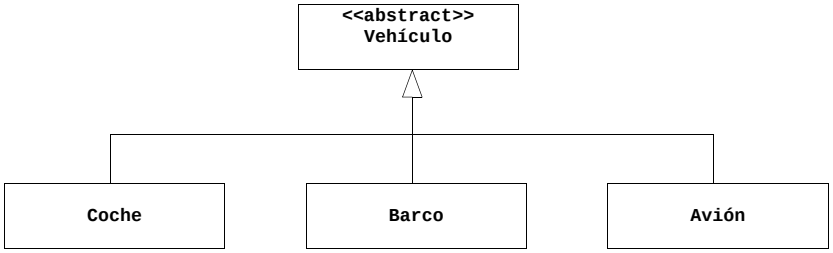
- **ambos conceptos son abstracciones de los objetos reales, tales como círculos, cuadrados, coches o aviones**
- **a ese tipo de clases las denominaremos *clases abstractas***







# Ejemplo: jerarquía de vehículos con clase abstracta










# Métodos abstractos

**Una clase abstracta puede tener *métodos abstractos***

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

**Permiten declarar en la superclase un comportamiento que deberán verificar todas sus subclases**

- **pero sin decir nada sobre su implementación**

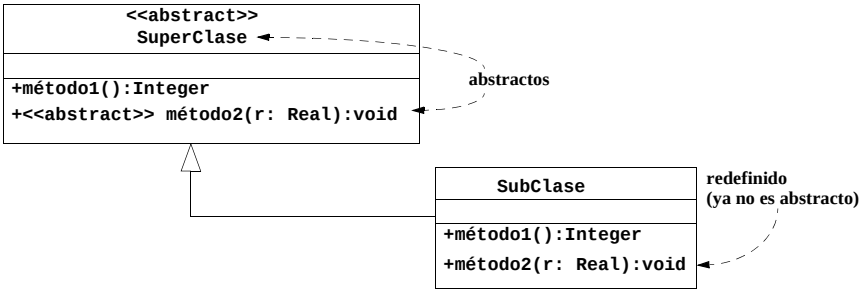
**Ejemplo de método abstracto en Java**

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

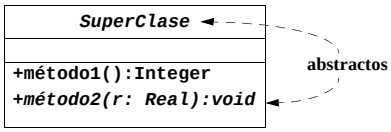
**no tiene cuerpo**

# Clases abstractas en diagramas de clases

**Se utiliza el estereotipo `<<abstract>>`:**




**También puede *escribirse en cursiva*:**

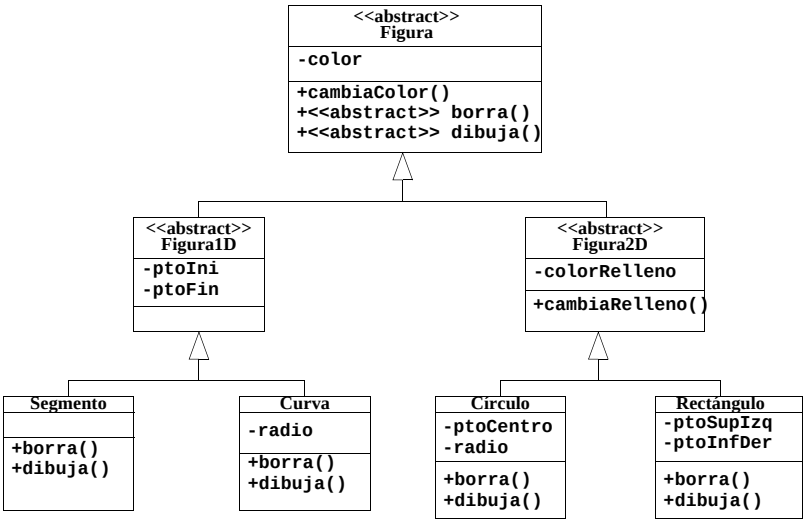









# Ejemplo: jerarquía de clases `Figura` y subclases










# 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 (hablaremos más de ello cuando veamos el polimorfismo)**

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

# Ejemplo: jerarquía de figuras

```
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();
}
```





```
public abstract class Figura1D extends Figura {

  // puntos de comienzo y final de la figura
  private final Punto ptoIni, ptoFin;

  /** Constructor … */
  public Figura1D(int color, Punto ptoIni,
                  Punto ptoFin) {
    super(color);
    this.ptoIni = ptoIni;
    this.ptoFin = ptoFin;){
  }

  // NO redefine ningún método abstracto

}
```





```
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
}
```

```
public class Recta extends Figura1D {

  /** Constructor … */
  public Recta(int color,
               Punto ptoIni, Punto ptoFin) {
    super(color, ptoIni, ptoFin);
  }

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

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





```
public class Círculo extends Figura2D {
  private Punto ptoCentro;
  private double radio;

  /** Constructor … */
  public Círculo(int color, int colorRelleno,
                 Punto ptoCentro, double radio){
    super(color, colorRelleno);
    this.ptoCentro = ptoCentro;
    this.radio = radio;
  }

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

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





# Ventajas y desventajas del uso de la herencia

**Ventajas:**

- **+ Mejora el *diseño***
  **Permite modelar relaciones de tipo “es un” que se dan en los problemas que se pretenden resolver**
- **+ Permite la *reutilización* del código**
  **Los métodos de la clase padre se reutilizan en las clases hijas**
- **+ Facilita la *extensión* de las aplicaciones**
  **Añadir una nueva subclase no requiere modificar ninguna otra clase de nuestro diseño**

**Principal desventaja:**

- **- Aumenta el *acoplamiento***
  **Las subclases están íntimamente acopladas con la superclase**

# 5.3 Polimorfismo

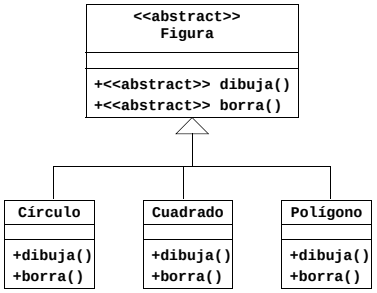
**Las *operaciones polimórficas* son aquellas que hacen funciones similares con objetos diferentes**

**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 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.**







# 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**

```
Vigilante v1=new Vigilante(...);
Vigilante v2=new VigilanteNocturno(...);
```

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

**`v1.sueldo()`** ← **usa el método `sueldo()` de la clase `Vigilante`, puesto que `v1` es un vigilante**

**`v2.sueldo()`** ← **usa el método `sueldo()` de la clase `VigilanteNocturno`, puesto que `v2` es un vigilante nocturno**

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:

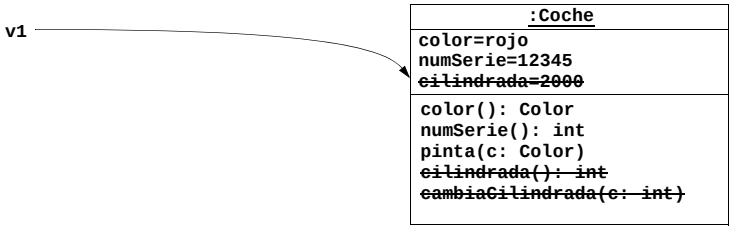
- un coche es un vehículo
  - cualquier operación de la clase `Vehículo` existe (sobrescrita o no) en la clase `Coche`
    `v.operaciónDeVehículo(...); // siempre correcto`
- un vehículo no es un coche
  - sería un error tratar de invocar la operación:
    `c.cilindrada(); // ERROR: cilindrada() no existe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





# Diferentes “puntos de vista”

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




Desde una referencia de tipo `Vehículo`, un coche se ve desde el punto de vista de un `Vehículo`

- Sólo se puede acceder a los atributos y métodos definidos en la clase `Vehículo`.

Desde una referencia de tipo `Coche` se podrían acceder a todos sus atributos y métodos.

# Conversión de referencias (*casting*)

Es posible convertir referencias

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

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 definidos en la clase `Vehículo`)
- a través de `c` le vemos como un `Coche` (y podemos invocar cualquiera de los métodos de esa clase y de sus superclases)

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







# Operador `instanceof`

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

Un *uso excesivo* del operador `instanceof` :

- *Elimina las ventajas* del polimorfismo
- Revela un *diseño incorrecto* de las jerarquías de clases

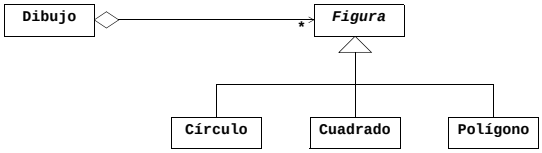






# Arrays de objetos de la misma jerarquía de clases

Gracias al polimorfismo[1] es posible que un array (o un `ArrayList`) contenga referencias a objetos de *distintas clases de una jerarquía*




La clase `Dibujo` contiene una colección de figuras (círculos, cuadrados y polígonos)

Si creamos la clase `Triángulo` que extiende a `Figura`

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

1. En particular a la regla 1 de la transparencia 42

# Ejemplo sencillo: 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();
}
```

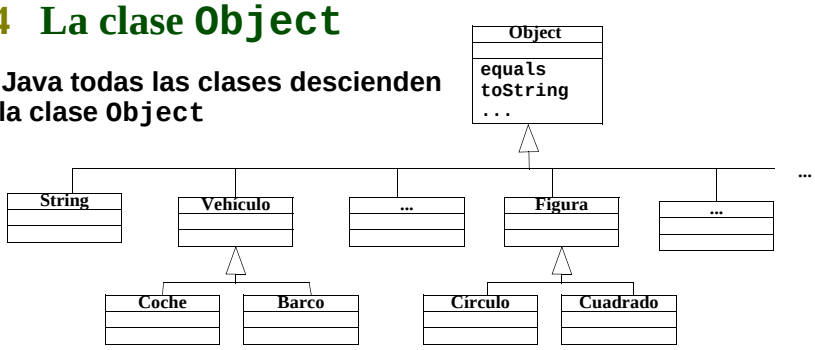
**Llama a la operación `borra` correspondiente a la clase del objeto**





# 5.4 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**

```
public class Clase {...}
```

**es transformado por el compilador en**

```
public class Clase extends Object {...}
```





# 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;
  }
}
```

aconsejable cuando se redefine un método para detectar errores

Redefine el método `equals` de la clase `Object`

Cambio de "punto de vista" para poder acceder a los campos `x` e `y` de `obj`





**Muchas clases estándar Java (p.e. `ArrayList`) 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**

¡Incorrecto!

```
public boolean equals(Coordenada obj){
  // ¡NO redefine el método equals
  // de la clase Object!
  ...
}
```





# Método `equals` y la clase `ArrayList`

**Varios métodos de la clase `ArrayList` utilizan el método `equals` del elemento para realizar búsquedas**

| Descripción | Interfaz |
|---|---|
| Busca el primer elemento de la lista igual a `ele` y lo elimina. Retorna `true` si ha eliminado el elemento | `boolean remove(E ele)` |
| Retorna `true` si la lista contiene algún elemento igual a `ele` | `boolean contains(Object ele)` |
| Retorna el índice de la primera aparición de un elemento igual a `ele`, o -1 si no hay ninguno | `int indexOf(Object ele)` |
| Retorna el índice de la última aparición de un elemento igual a `ele`, o -1 si no hay ninguno | `int lastIndexOf(Object ele)` |

# Método `toString`

**Se encuentra definido en la clase `Object` como:**

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

- **es utilizado cuando se concatena un objeto con un string**
- **por defecto retorna un string con el nombre de la clase y la dirección de memoria que ocupa el objeto**

```
println("Coord: " + c);
```

**Salida por consola**

```
Coord: 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 por consola:**

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





# Método `toString` de las *Java Collections*

**Las *Java Collections* (`ArrayList`, `LinkedList`, `HashMap`, ...) redefinen el método `toString` de `Object`**

- **el método `toString` muestra la colección con el formato**

  **`[elemento 0, elemento 1, …, elemento n-1]`**
- **a su vez cada elemento se muestra invocando su propio método `toString`**

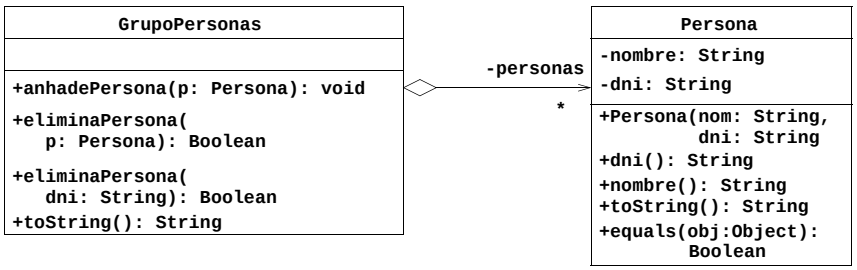
**También existe un método `toString` para los *arrays***

- **Definido en `java.util.Arrays`:**

```
int[] a = {4, 30, -5, 0, 23, -7};
System.out.println("a:" + Arrays.toString(a));
// muestra "a:[4, 30, -5, 0, 23, -7]"
```

- **En `java.util.Arrays` existen más métodos para trabajar con arrays: `sort`, `fill`, `copyOf`, `equals`, ...**

# Ejemplo `ArrayList`, `equals` y `toString`




```java
/**
 * Persona (clase muy sencilla para este ejemplo)
 * @author Métodos de Programación (UC)
 * @version curso 19-20
 */
public class Persona {

   // datos: nombre y dni
   private final String nombre;
   private final String dni;  // único para cada persona
```





Ejemplo ArrayList, `equals` y `toString` (cont.)

```java
   /** Construye una persona con los datos indicados ... */
   public Persona(String nombre, String dni) {
      this.nombre = nombre;
      this.dni = dni;
   }
   /** Retorna el dni de la persona ... */
   public String dni() {
      return dni;
   }
   /** Retorna el nombre de la persona ... */
   public String nombre() {
      return nombre;
   }
   @Override
   public String toString() {
      return "(dni=" + dni + ", nombre=" + nombre + ")";
   }
   @Override
   public boolean equals(Object obj) {
      return dni.equals(((Persona) obj).dni);
   }
```

**Para los métodos sobrescritos no es necesario poner comentarios de documentación.**





Ejemplo ArrayList, `equals` y `toString` (cont.)

```java
import java.util.ArrayList;
/**
 * Un grupo de personas
 * @author MP
 * @version 1.0
 */
public class GrupoPersonas {
   // personas pertenecientes al conjunto
   private ArrayList<Persona> personas = new ArrayList<Persona>();

   public static class PersonaYaExistente extends RuntimeException {}
   public static class DniNoExistente extends RuntimeException { }
   public static class PersonaNoEnGrupo extends RuntimeException { }

   /**
    * Añade una persona al grupo.
    * @param p persona a añadir
    * @throws PersonaYaExistente si la persona ya está en el grupo
    */
   public void anhadePersona(Persona p) throws PersonaYaExistente {
      if (personas.contains(p)) {
         throw new PersonaYaExistente();
      }
      // añade la persona
      personas.add(p);
   }
```

**Para que `contains()` funcione correctamente es necesario que la clase `Persona` haya redefinido su método `equals()`**

Ejemplo ArrayList, `equals` y `toString` (cont.)

```
    /**
     * Elimina una persona del grupo.
     * @param p persona a eliminar
     * @throws PersonaNoEnGrupo si la persona no está en el grupo
     */
    public void eliminaPersona(Persona p) throws PersonaNoEnGrupo {
       boolean eliminado = personas.remove(p);
       if (!eliminado) {
          throw new PersonaNoEnGrupo();
       }
    }

    /**
     * Elimina del grupo la persona con el DNI indicado
     * @param dni DNI de la persona a eliminar
     * @throws DniNoExistente si no existe ninguna persona en el
     * grupo con ese DNI.
     */
    public void eliminaPersonaConDNI(String dni)throws DniNoExistente{
       Persona persona = buscaPersona(dni);
       if (persona == null) {
          throw new DniNoExistente();
       }
       personas.remove(persona);
    }
```

**Para que `remove()` funcione correctamente es necesario que la clase `Persona` haya redefinido su método `equals()`**





Ejemplo ArrayList, `equals` y `toString` (cont.)

```
    /**
     * Busca la persona con el DNI indicado en el ArrayList personas.
     * @param dni DNI de la persona buscada.
     * @return persona con el DNI indicado o null si no hay ninguna
     * persona con ese DNI.
     */
    public int buscaPersona(String dni) {
       for (Persona p: personas) {
          if (p.dni().equals(dni)) {
             return p;
          }
       }
       return null;
    }

    @Override
    public String toString() {
       return personas.toString();
    }

 }
```

**Para los métodos sobrescritos no es necesario poner comentario de documentación.**





# 5.5 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”, 4ª edición. Addison-Wesley, 2005.**
- **Deitel, Harvey M. y Deitel, Paul J., “Cómo programar en Java”, 9ª edición. Pearson Educación, México, 2012.**