



En este caso se entiende que los paquetes se deben de buscar en esas tres rutas raíz

Así para el paquete **prueba.reloj** tiene que haber una carpeta **prueba**, dentro de la cual habrá una carpeta **reloj**. La carpeta que contiene a **prueba** tiene que formar parte del **classpath**.

Una clase se debe declarar como perteneciente a su paquete usando la instrucción **package** al principio del código (sin usar esta instrucción, la clase no se puede compilar). Una clase que requiere la instrucción **package** tiene que colocarla como primera instrucción del código:

```
//Clase perteneciente al paquete tema5 que está dentro del
//paquete ejemplos
package ejemplos.tema5;
```

Y eso significará que esa clase debe de estar físicamente almacenada en un directorio llamado **tema5** que estará dentro del directorio **ejemplos** que a su vez es parte del **classpath**.

Al compilar el programa mediante el comando **javac**, todas las clases utilizadas deben estar compiladas y colocadas correctamente en el **classpath**. O bien se puede utilizar el parámetro **-cp** para indicar una ruta que debe ser considerada como raíz de paquetes necesarios. Ejemplo:

```
javac -cp "d:\utiles\ejemplo16" Reloj.java
```

Esto compila la clase **Reloj** pero indicando que la ruta **d:\utiles\ejemplo16** contiene paquetes necesarios para la compilación. Con lo cual en la carpeta **Reloj.java**

A la hora de lanzar el programa debemos colocarnos en la carpeta raíz de los paquetes que contienen la clase a compilar. Suponiendo que **Reloj** es una clase del paquete **net.jorgesanchez.utiles** entonces la compilación sería:

```
java net.jorgesanchez.utiles.Reloj
```

El programa **java** lo que hace es cambiar todos los puntos por símbolos \ ó / dependiendo del sistema operativo en el que estemos. y busca el archivo **Reloj.class** en esa ruta añadida a todas las que tiene en el **classpath**. Si lo encuentra la ejecuta. El programa **java** también admite el parámetro **-cp** para añadir una ruta que no esté en el **classpath** (y que contenga paquetes que utilice nuestro programa).

En los entornos de desarrollo o IDEs (**NetBeans**, **Eclipse**,...) se puede uno despreocupar de la variable **classpath** ya que los propios IDE se encargan de gestionarla .

Convenciones en la definición de paquetes Java

Los siguientes son convenciones o estándares acordados en la definición de paquetes en Java:

- El nombre del paquete se define de manera inversa al dominio de la organización o grupo. Por ejemplo, dominioempresa.com puede ser usado como nombre de paquete así: com.dominio_empresa.utilidades.
- El nombre del paquete debería definirse en minúscula. Si existe varias palabras en el nombre se pueden separar con guión bajo (_).

Uso de paquetes

En los ficheros de código Java se usa la palabra reservada package para especificar a qué paquete pertenecen. Suele indicarse como primera sentencia:

```
package java.awt.event;
```

Para usar un paquete dentro del código se usa la declaración import. Si sólo se indica el nombre del paquete:

```
import java.awt.event.*;
```

se importan todas las clases que contiene. Si además del nombre del paquete se especifica una clase, sólo se importa esa clase:

```
import java.awt.event.ActionEvent;
```

Después de añadir alguna de estas sentencias, se puede hacer referencia a la clase ActionEvent usando su nombre:

```
ActionEvent myEvent = new ActionEvent();
```

Si no se hubiera importado la clase o el paquete, cada vez que tuviéramos que usarla habría que especificarla por su **fully qualified class name**, que no es más que el nombre del paquete seguido por el nombre de la clase:

```
java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();
```

Si lo que se desea es importar todos los miembros estáticos de una clase, note la sentencia static después de import. (A partir de J2SE 5.0 en adelante)

```
import static java.awt.Color.*;
```

Si lo que se desea es importar un sólo miembro estático de una clase. (A partir de J2SE 5.0 en adelante)

```
import static java.awt.color.ColorSpace.CS_GRAY*;
```



todas las clases que no tengan un paquete declarado. Aunque esto no genera errores de compilación ni nada parecido, siempre es recomendable declarar un paquete a cada componente de nuestro programa Java para poder darle diferentes niveles de seguridad o acceso a dichos componentes y mantener todo ordenado.

- **Al momento de declarar el paquete en Java:**

Es común usar la primera letra en mayúscula cuando se declara una clase, pues bien, cuando se declaran paquetes es común que todas las letras estén en minúscula y en caso de ser varias palabras separarlas por un guion bajo "_" por ejemplo "mi_paquete" es adecuado mientras que "MiPaquete" aunque no es incorrecto, no es una buena práctica.

En estos momentos ya tenemos claro qué es y para qué sirve un paquete en Java, también sabemos cómo se declaran los paquetes en Java y probablemente tendremos una noción de los cambios que estos generan al interior de la estructura de nuestro proyecto. A continuación, veremos un ejemplo que consistirá de cuatro clases diferentes en ubicaciones (paquetes) también diferentes y luego veremos cómo queda la estructura de nuestro proyecto

Ejemplo de paquetes en Java

A continuación pondré la declaración de cuatro clases diferentes en paquetes diferentes, cada clase se llamará "Clase_1", "Clase_2", "Clase_3" y "Clase_4" respectivamente. Voy a jugar un poco con las carpetas donde quedará alojada cada una de las clases para que afiances un poco más el concepto y comprendas bien el funcionamiento de los paquetes en Java y quizás soluciones alguna duda. Por tal motivo, te invito a intentar descifrar la ubicación exacta de cada clase al interior del proyecto según la declaración de su respectivo paquete y ver que sí hayas entendido adecuadamente el tema. Veamos:

- **Clase número uno:**

```
package mis_clases.clases_publicas.clase_1;
public class Clase_1
{
}
```

- **Clase número dos:**

```
package mis_clases.clase_2;
class Clase_2
{
}
```

- **Clase número tres:**

```
package mis_clases;
class Clase_3
{
}
```

- **Clase número cuatro:**

```
class Clase_4
{
}
```

En la imagen de abajo se puede apreciar la estructura final obtenida tras la declaración de los paquetes de cada una de las cuatro clases de Java. Veamos con detalle esto para quedar claros.





Ejercicio uº 40

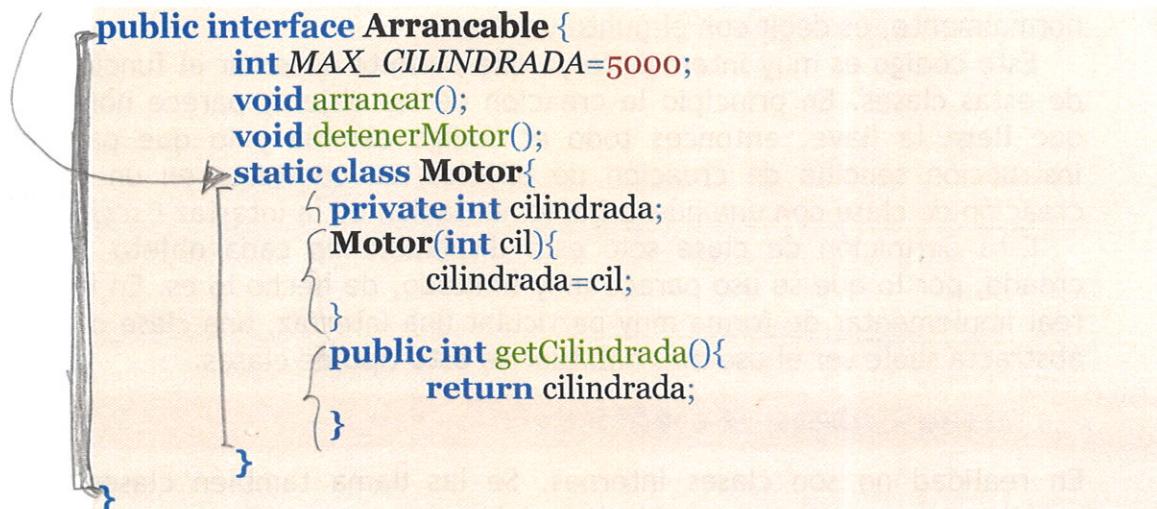
- Sopa de letras. Matriz 10x10. 4 Palabras

fundamentos de programación
(unidad 7) relaciones entre clases. herencia

El objeto **m** se puede utilizar sin ningún problema aunque no haya un coche definido.

En las clases estáticas no se puede acceder a ningún elemento de la clase contenedora. Además estas sí pueden tener métodos y propiedades estáticas.

Una capacidad muy interesante es las clases estáticas pueden ser internas a una interfaz y esto sí proporciona funcionalidades muy interesantes. Ejemplo:



```
public interface Arrancable {
    int MAX_CILINDRADA=5000;
    void arrancar();
    void detenerMotor();
    static class Motor{
        private int cilindrada;
        Motor(int cil){
            cilindrada=cil;
        }
        public int getCilindrada(){
            return cilindrada;
        }
    }
}
```

Para crear un objeto de la clase interna motor valdría con:

```
Arrancable.Motor m=new Arrancable.Motor(650);
```

(7.9) creación de paquetes

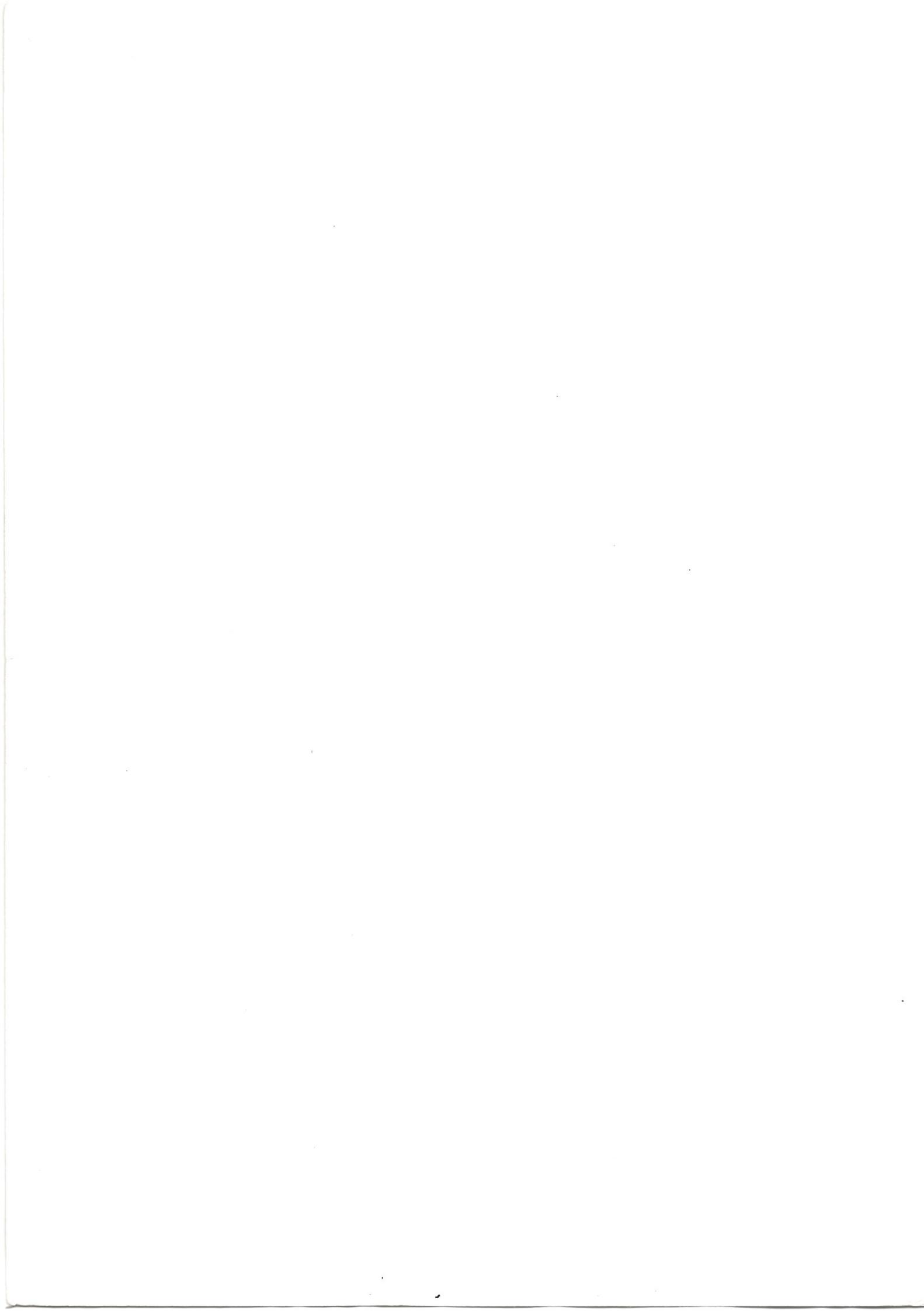
En Java las aplicaciones se organizan en paquetes. Los paquetes son contenedores en los que se almacenan las clases e interfaces. Todas las clases creadas y utilizadas deben residir en un paquete.

El compilador de Java usa los paquetes para organizar la compilación y ejecución. Es decir, podemos entender que un paquete es una biblioteca en la que se almacenan las clases. De hecho el nombre completo de una clase es el nombre del paquete en el que está la clase seguido de un punto y luego el nombre de la clase. Es decir si la clase **Coche** está dentro del paquete **locomoción**, el nombre completo de Coche es **locomoción.Coche**.

A veces resulta que un paquete está dentro de otro paquete, entonces habrá que indicar la ruta completa a la clase. Por ejemplo **locomoción.motor.Coche**

Mediante el comando **import** (visto anteriormente), se evita tener que colocar el nombre completo. El comando **import** se coloca antes de definir la clase. Ejemplo:

```
import locomoción.motor.Coche;
```





Lo curioso es que nada más acabar la llave se utiliza el propio método definido ya que al cerrar la llave volvemos a la instrucción de creación y como tenemos el nuevo objeto, podemos utilizar el método enviar.

Sin duda es una forma enrevesada de trabajar, pero en realidad las clases anónimas se utilizan mucho en Java (aunque no de esa forma), especialmente al crear interfaces gráficas.

Es mucho más habitual utilizarle para dar forma temporal a una implementación temporal de trabajarles de esta forma:

```
public interface Escribible {  
    public void escribir(String texto);  
}
```

Esta interfaz como se puede observar simplemente define un método escribir que recibe un texto como parámetro. A través de esta interfaz podemos crear objetos de clases anónimas, por ejemplo:

```
public class ClaseAnonima {  
    public static void main(String[] args) {  
        int i;  
        Escribible msg=new Escribible(){  
            @Override  
            public void escribir(String texto) {  
                System.out.println(texto);  
            }  
        };  
        Escribible msg2=new Escribible(){  
            @Override  
            public void escribir(String texto) {  
                JOptionPane.showMessageDialog(null, texto);  
            }  
        };  
        do {  
            i=Integer  
                .parseInt(JOptionPane  
                    .showInputDialog("1) Escribir en la consola\n"  
                        +"2) Escribir en una pantalla gráfica\n"  
                        +"3) Salir"));  
            if(i==1){  
                msg.escribir("Hola");  
            }  
            else if(i==2){  
                msg2.escribir("Hola");  
            }  
        } while (i!=3);  
    }  
}
```

Otra razón para utilizar esta técnica es para conseguir implementaciones especiales de clases o interfaces adaptadas a las necesidades de un método concreto.

Las clases internas a un método se usan igual que las clases normales, pero simplemente su rango de acción se reduce al tamaño del método en el que se declaran.

(7.8.5) Clases internas dentro de un bloque

Las clases internas más limitadas en cuanto a su uso son las que se definen en un bloque de código concreto. Un ejemplo:

```
public static void main(String[] args) {
    int i;
    do{
        i=Integer.parseInt(
            JOptionPane.showInputDialog(
                "Elija entre 1) Mandar un mensaje o 2) Salir"));
        if(i==1){
            class Msg extends Socket {
                int mensaje;
                Msg(int mensaje) throws IOException{
                    super("localhost",1200);
                    this.mensaje=mensaje;
                }
                public void enviar() throws IOException {
                    super.sendUrgentData(mensaje);
                }
            }
            try{
                Msg m=new Msg(25);
                m.enviar();
            }
            catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Error");
            }
        }//fin del if
    }while(i!=2);
    Msg m2=new Msg(); //ERROR!!! La clase Msg ya no está
                      //disponible
}
```

En el ejemplo, la clase **Msg** se crea dentro del **if**. Se trata de una clase que hereda de la clase **Socket**, métodos para conectar y enviar información a través de la red. A pesar de la dificultad de entender el código a estas alturas debido al manejo de errores (instrucciones **throws**, **try** y **catch** que se verán en el tema siguiente) y a que la clase **Socket** es compleja.

El objeto m es de tipo **Coche.Motor** y aunque la creación en una clase diferente a la contenedora es compleja, es posible (siempre que el acceso sea público). Otra cosa es si esto es interesante o no.

(7.8.3) acceso a propiedades

Otra cuestión es el hecho de que las clases internas pueden acceder a propiedades de la clase contenedora:

```
public class Coche {  
    public int velocidad;  
    public Motor motor;  
  
    public Coche(int cil) {  
        motor=new Motor(cil);  
        velocidad=0;  
    }  
  
    public class Motor{ //Clase interna  
        public int cilindrada;  
        public Motor(int cil){  
            cilindrada=cil;  
        }  
  
        public void arrancar(){  
            velocidad=1;  
        }  
    }  
}
```

El método arrancar pone la propiedad **velocidad** a 1, lo sorprendente es que esta propiedad es de la clase **Coche** y no de **Motor**; y el código funciona correctamente. Lo que significa es que sin problemas podemos acceder a las propiedades de la clase contendora.

Otro problema está en el operador **this**. El problema es que al usar **this** dentro de una clase interna, éste se refiere al objeto de la clase interna (es decir **this** dentro de **Motor** se refiere al objeto **Motor**). Para poder referirse al objeto contenedor (al coche) se usa **Clase.this** (por ejemplo: **Coche.this**). Ejemplo:

```
public class Coche {  
    public int velocidad;  
    public int cilindrada;  
    public Motor motor;  
  
    public Coche(int cil) {  
        motor=new Motor(cil);  
        velocidad=0;  
    }  
}
```

(7.7.5) lista completa de métodos de la clase Object

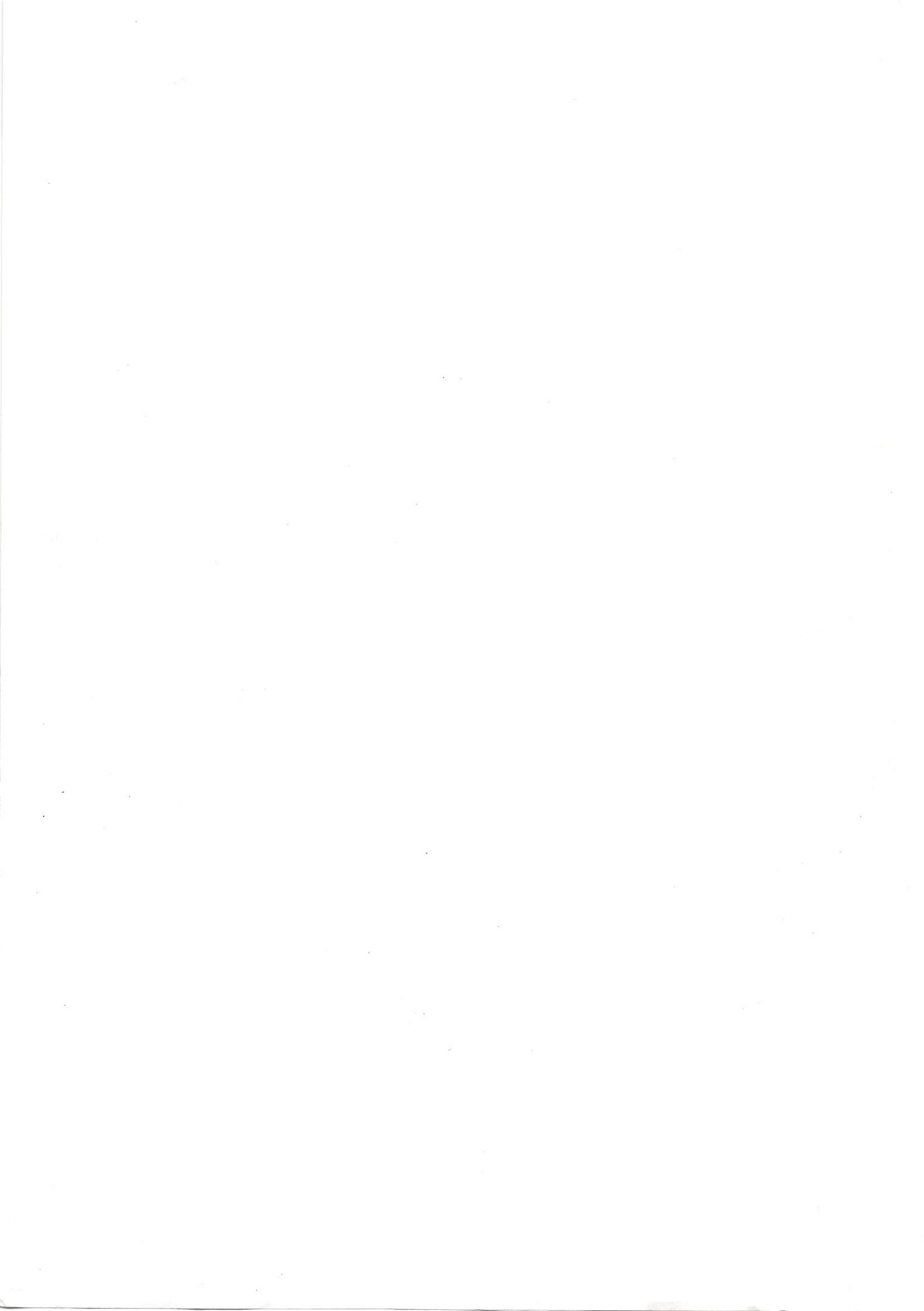
método	significado
<code>protected Object clone()</code>	Devuelve como resultado una copia del objeto.
<code>boolean equals(Object obj)</code>	Compara el objeto con un segundo objeto que es pasado como referencia (el objeto <code>obj</code>). Devuelve <code>true</code> si son iguales.
<code>protected void finalize()</code>	Destructor del objeto
<code>Class getClass()</code>	Proporciona la clase del objeto. Permite saber el nombre de la clase
<code>int hashCode()</code>	Devuelve un valor <code>hashCode</code> para el objeto
<code>void notify()</code>	Activa un hilo (thread) sencillo en espera.
<code>void notifyAll()</code>	Activa todos los hilos en espera.
<code>String toString()</code>	Devuelve una cadena de texto que representa al objeto
<code>void wait()</code>	Hace que el hilo actual espere hasta la siguiente notificación
<code>void wait(long tiempo)</code>	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo
<code>void wait(long tiempo, int nanos)</code>	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo o hasta que otro hilo interrumpa al actual

(7.8) clases internas

(7.8.1) uso de clases internas

Se llaman clases internas a las clases que se definen dentro de otra clase. Requiere esta técnica una mayor pericia por parte del programador. Y los beneficios tampoco son tantos, entre ellos:

- ◆ Acceder a los métodos y propiedades que posee la clase contenedora
- ◆ Definir objetos cuya existencia está totalmente supeditada a objetos de la clase contenedora
- ◆ Aplicar a clases que se desea sean ocultas al resto
- ◆ Usar clases anónimas para hacer definiciones sobre necesidades en ejecución
- ◆ Casi imprescindibles para manejar eventos



Es necesario el uso de `instanceOf` ya que `equals` puede recoger cualquier objeto `Object`. Para que la comparación sea válida primero hay que verificar que el objeto es un coche. El argumento o siempre hay que convertirlo al tipo `Coche` para utilizar sus propiedades de `Coche`.

(7.7.2) código hash

El método `hashCode()` permite obtener un número entero llamado **código hash**. Este código es un entero único para cada objeto que se genera aleatoriamente según su contenido. No se suele redefinir salvo que se quiera anularle para modificar su función y generar códigos hash según se desee.

(7.7.3) clonar objetos

El método `clone` está pensado para conseguir una copia idéntica de un objeto. Es un método `protected` por lo que sólo podrá ser usado por la propia clase y sus descendientes, salvo que se le redefina con `public`. La copia realizada por `clone` es un nuevo objeto y por lo tanto tendrá una nueva referencia; es decir modificar el objeto clonado no afecta al original.

Es lógico no poder utilizar el método `clone` fuera de la clase salvo redefinirle como `public`, la razón estriba en que solo la propia clase puede saber como clonar correctamente un objeto. El método `clone` en la clase `Object` duplica literalmente todas las propiedades; cuando estas son tipos primitivos no hay problemas, pero cuando son referencias a objetos, entonces el clonado fallará porque no duplica los objetos, sino que duplicará sólo las referencias.

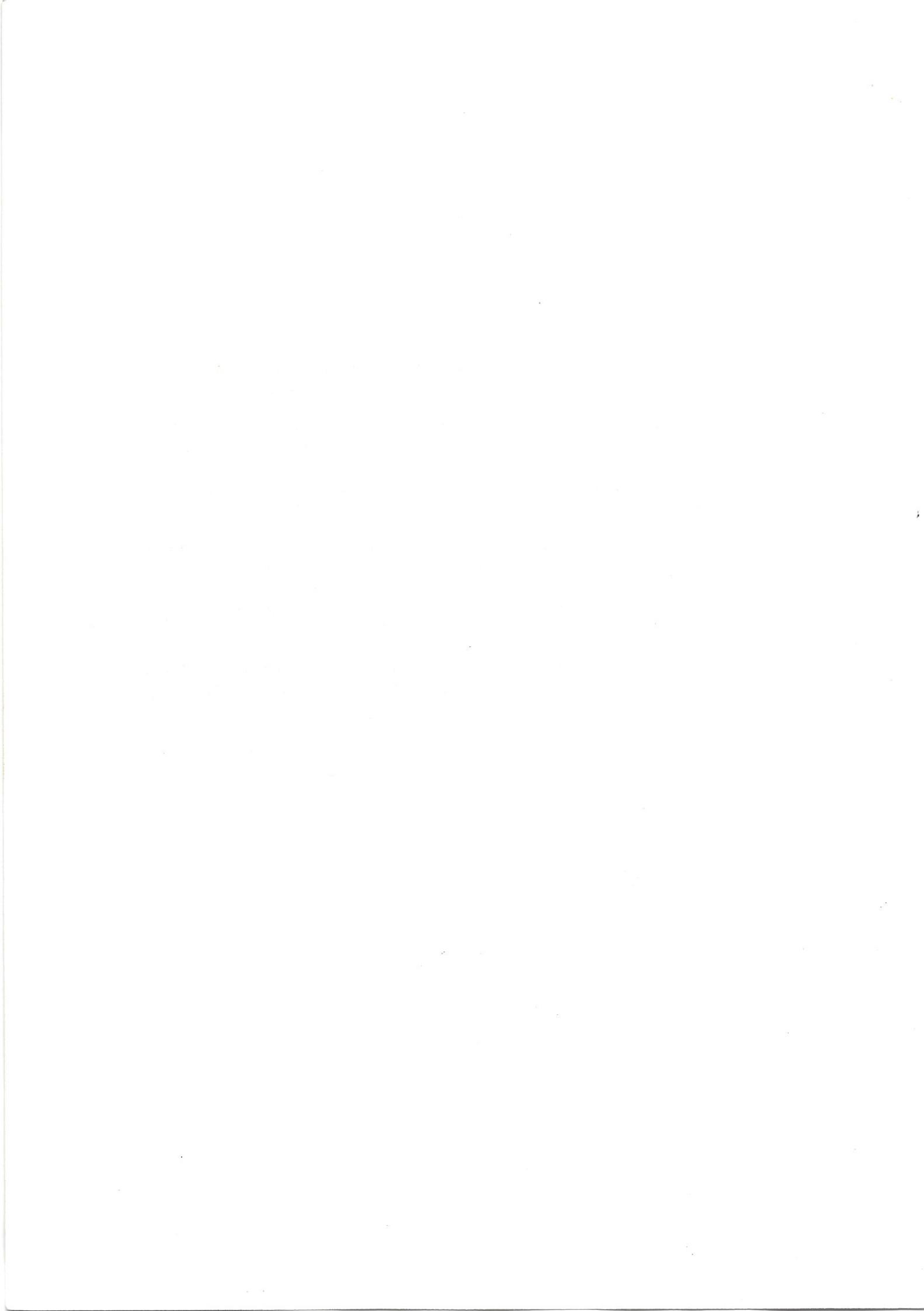
Para poder redefinir el método `clone`, la clase debe implementar la interfaz `Cloneable` (pertenece al paquete `java.lang`), que no contiene ningún método pero sin ser incluida al usar `clone` ocurriría una excepción del tipo `CloneNotSupportedException`. Esta interfaz es la que permite que el objeto sea clonable.

De ese modo podremos saber en tiempo de ejecución si una clase puede ser clonada mediante el operador `instanceOf`:

```
if(obj instanceof Cloneable){...}
```

Ejemplo:

```
public class Coche extends Vehiculo implements Arrancable,  
Cloneable{  
    public Object clone(){  
        try{  
            return (super.clone());  
        }catch(CloneNotSupportedException cnse){  
            return null;  
        }  
    }  
    ....
```





```

public class Prueba {
    Escribible escritor;
    public Prueba(Escribible e){
        escritor=e;
    }
    public void enviaTexto(String s){
        escritor.escribe(s);
    }
}

```

En el ejemplo *escritor* es una variable de la interfaz *Escribible*, cuando se llama a su método *escribe*, entonces se usa la implementación de la clase *Texto*. La ventaja, no necesitamos saber a qué clase pertenece el escritor puesto que hará referencia a un objeto concreto, se usará el método *escribe* de ese objeto.

(7.7) La clase Object

Todas las clases de Java poseen una superclase común, esa es la clase **Object**. Al ser superclase de todas las clases de Java, todos los objetos Java en definitiva son de tipo **Object**, lo que permite crear métodos genéricos. Ejemplo:

```

public class C{
    public static void f(Object o){
        ...
    }
    ...
}
public class D{
    public static void main(String args[]){
        //todas las líneas son válidas
        C.f(new String());
        C.f(new int[7]);
        C.f(new C());
        ...
    }
}

```

Al método estático *f* se le puede enviar cualquier clase de objeto. Para que funcione correctamente el código interior debe hacerse con habilidad.

Otra ventaja es que **Object** posee una serie de métodos muy interesantes que todas las clases heredan. Pero, normalmente, hay que redefinirlos para que funcionen adecuadamente adaptándolos a la clase correspondiente.

Es decir, **Object** proporciona métodos que son heredados por todas las clase. La idea es que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes como comparar, clonar, escribir,....

relación sino que hay una **obligación** por parte de la clase que implemente la interfaz de redefinir los métodos de ésta.

Una interfaz se crea exactamente igual que una clase (se crean en archivos propios también), la diferencia es que la palabra **interface** sustituye a la palabra **class** y que **sólo se pueden definir en un interfaz propiedades y métodos abstractos**.

Todas las interfaces son abstractas y sus métodos también son todos abstractos y públicos (no hace falta poner el modificador **abstract**, se toma de manera implícita). Las variables se tienen obligatoriamente que inicializar. Ejemplo:

```
public interface Arrancable {  
    int MAX_CILINDRADA=5000;  
    void arrancar();  
    void detenerMotor();  
}
```

Los métodos son simples prototipos y toda variable se considera una **constante** estática (a no ser que se redefina en una clase que implemente esta interfaz, lo cual no tendría mucho sentido).

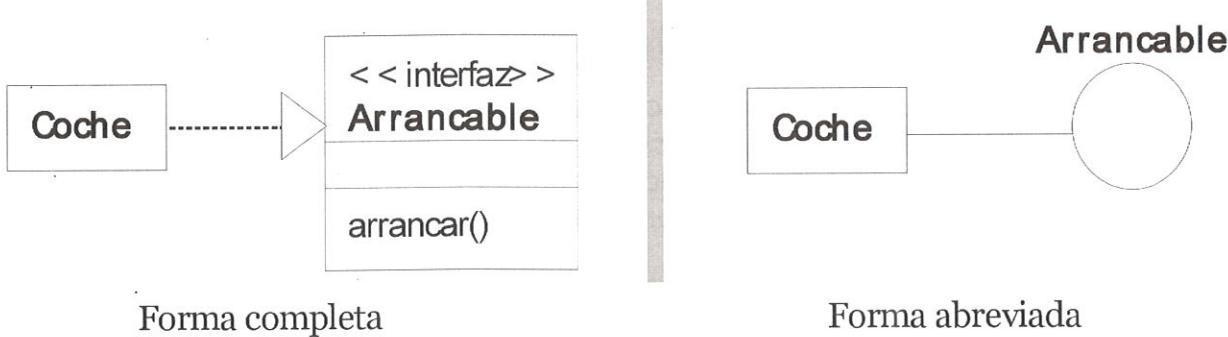


Ilustración 7-9, Diagramas de clases UML sobre la interfaz Arrancable

(7.6.3) subinterfaces

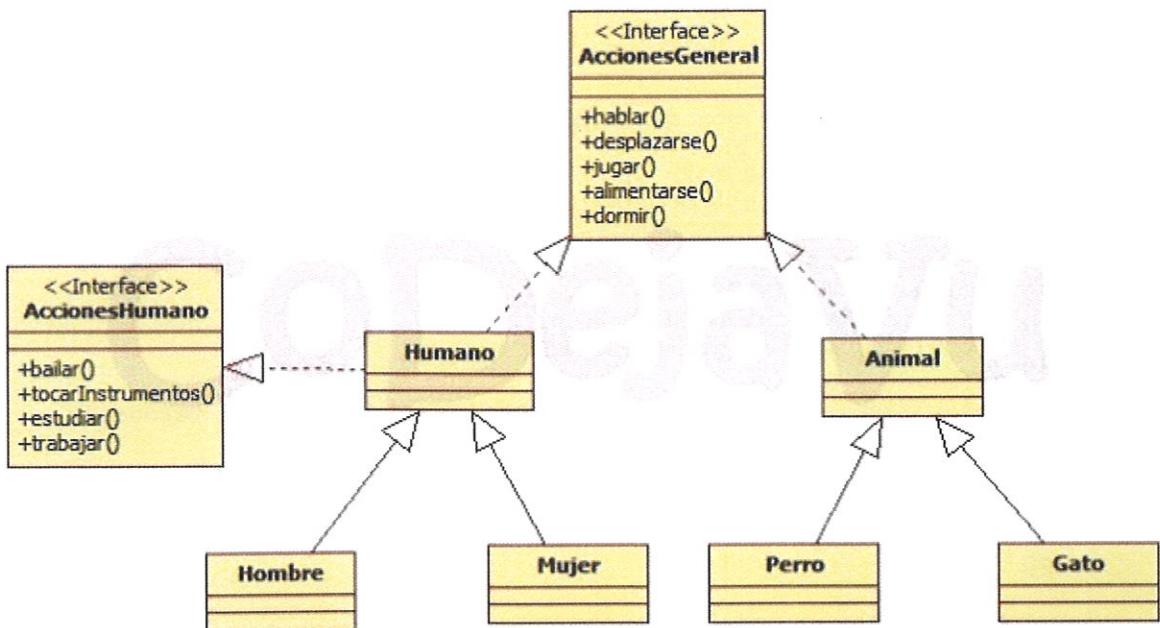
Una interfaz puede heredarse de otra interfaz, como por ejemplo en:

```
interface Dibujable extends Escribible, Pintable {  
    ...
```

Dibujable es subinterfaz de **Escribible** y **Pintable**. Es curioso, pero los interfaces sí admiten herencia múltiple (la razón es que no existen los problemas de C++ ya que los interfaces no generan objetos creados en memoria hasta que se implementen en una clase concreta). Esto significa que la clase que implemente el interfaz **Dibujable** deberá incorporar los métodos definidos en **Escribible** y **Pintable**.

con clases abstractas nos aseguramos que los métodos y atributos solo están disponibles para las clases que las implementen...

Veamos un Ejemplo.



En el diagrama de clases vemos 6 **clases concretas** y 2 **interfaces**, las clases **Humano** y **Animal** son clases **padre** de "**Hombre y Mujer**" y "**Perro y Gato**" respectivamente, ahora bien, **Humano** y **Animal** son clases diferentes con un árbol de **herencia** marcado, pero ambas poseen características comunes que podemos usar por medio de la interface **AccionesGeneral**.

Podemos decir que tanto un **Hombre** como un **Gato** pueden caminar, usando para esto el método **desplazarse()**, donde cada clase dará el mecanismo de desplazamiento, por ejemplo el hombre lo hace en 2 piernas mientras que el gato en 4 patas (o dependiendo de la forma como lo realicen), y este mismo concepto puede aplicarse a los otros métodos enmarcados en la Interface **AccionGeneral** que tanto Humanos como Animales comparten.

También tenemos la **interface AccionesHumano** para esos métodos o características que solo son aplicables a los humanos y que tanto **Hombre** como **Mujer** pueden adoptar, así la clase **Humano** podrá simular la herencia múltiple al implementar las 2 interfaces mencionadas.

```

9 public class Principal implements
10 InterfacePrincipal, otraInterface, otraMas{
11     public void metodoAbstracto() {
12         /**Implementación definida por la clase concreta*/
13     }
14
15     public String otroMetodoAbstracto() {
16         /**Implementación definida por la clase concreta*/
17         return "retorno";
18     }
19
20     public void metodoAbstractoDeOtraInterface() {
21         /**Implementación definida por la clase concreta*/
22     }
23     public void metodoAbstractoDeOtraMas() {
24         /**Implementación definida por la clase concreta*/
25     }
26}
27

```

Vemos que la clase Principal implementa 3 interfaces, por eso decimos que simulan la herencia múltiple ya que heredamos los métodos que nos brindan.....veamos en general las principales características.

Características de las Interfaces.

- Todos los métodos de una interfaz son implícitamente **public abstract**, no es necesario especificarlo en la declaración del mismo.
- Todas las variables y atributos de una interfaz son implícitamente constantes (**public static final**), no es necesario especificarlo en la declaración del misma
- Los métodos de una interfaz no pueden ser: **static, final, strictfp ni native**.
- Una interfaz puede heredar (**extends**) de una o más interfaces.
- Una interfaz no puede heredar de otro elemento que no sea una interfaz.
- Una interfaz no puede implementar (**implements**) otra interfaz.
- Una interfaz debe ser declarada con la palabra clave **interface**.
- Los tipos de las interfaces pueden ser utilizados polimórficamente.
- Una interfaz puede ser **public** o **package** (valor por defecto).
- Los métodos toman como ámbito el que contiene la interfaz.





- ◆ En los parámetros de un método. En ese caso indica que el valor recibido es una constante que no puede cambiar de valor dentro del método

```
public class Mate {  
    public static int[] arrayAleatorio(final int tam){  
        int array[] = new int[tam];  
        tam++; //error no se puede cambiar tam  
        for(int i=0; i<array.length; i++){  
            array[i] = (int)(Math.random()*10+1);  
        }  
        return array;  
    }  
}
```

Se utiliza sobre todo para marcar que un determinado valor no se debe cambiar en el método. En el ejemplo la instrucción remarcada es absurda, la función devuelve un array aleatorio con tantos elementos como valor tenga *tam*, lo lógico es que *tam* sea de sólo lectura.

- ◆ En la declaración de un método, indica que dicho método no puede ser sobrescrito en una subclase. Es decir si en la clase *Vehiculo*, el método *parar* se define con la palabra *final* (por ejemplo `public final void parar(...)`) entonces la clase *Coche* (heredera de *Vehiculo*) no puede redefinir el método. Si una subclase intentar sobrescribir el método, el compilador de Java avisará del error.

Una ventaja de estos métodos es que invocarlos es más rápido porque al saber el compilador que no pueden ser sobrescritos, les coloca como definitivos acelerando su ejecución. Los métodos privados se toman como constantes ya que no pueden ser redefinidos.

No obstante en la práctica no se usan mucho ya que no se puede saber si se va a necesitar redefinir el método o no en el futuro.

- ◆ Si se utiliza al definir una clase (por ejemplo `public final class Coche`) significará que esa clase no puede tener descendencia. Al igual que ocurre con un método de tipo final, hay que tener precaución porque es difícil saber si en el futuro necesitaremos o no hacer descendientes de una clase.

Ejercicio nº 34

- Dibuja adjunta

Ejercicio nº 35

- Crear la clase final Matiz con métodos estáticos

(22) máximos, mínimos, sumatoria y media que amplie la funcionalidad de la clase *Math*. Los métodos recogerán un array de números double y devolverán el resultado. Crear un método main para pruebas.

Ejemplo:

```
public abstract class Vehiculo {  
    public int velocidad=0;  
    abstract public void acelera();  
    public void para() {  
        velocidad=0;  
    }  
}  
  
public class Coche extends Vehiculo {  
    public void acelera() { //obligatoriamente hay que definirle  
        velocidad+=5;  
    }  
}  
  
public class Prueba {  
    public static void main(String[] args) {  
        Coche c1=new Coche();  
        c1.acelera();  
        System.out.println(c1.velocidad);  
        c1.para();  
        System.out.println(c1.velocidad);  
    }  
}
```

La diferencia con las interfaces (comentadas más adelante) estriba en que las clases abstractas pueden combinar métodos abstractos sin definir con métodos definidos (por lo tanto no abstractos). En las interfaces todos los métodos son abstractos.

En UML las clases abstractas aparece con el nombre en cursiva. Los métodos abstractos también aparecerán en cursiva:

<i>Vehiculo</i>
+velocidad:int=0
+acelera() +para()

```
2. ((Sum) sum).sumAbsolute();
```

El compilador nos permite pasar, pero al momento de ejecutar el código tendremos el error de ClassCastException.

Result: 5.0

Result: 8.0

En éste ejemplo ejecutamos 2 veces el metodo doCalculate con 2 instancias de objeto de diferente tipo, pero que heredan de un tipo común, y como el método espera un tipo MathOperator se realiza el casting de las variables de referencia sum3 y m1 y si bien apenas el metodo tiene una sola línea de código, en la vida real mas allá de ésta didáctica puede ser un proceso con mucha lógica, el cual lo reutilizamos sólo con el hecho de aprovechar las bondades del polimorfismo.

Podemos afinar las conversiones entre tipos si hacemos uso de la palabra reservada instanceof para garantizarnos a nosotros mismos que vamos a hacer un casting válido, por ejemplo:

```
view source
print?
1.MathOperator sum2 = new MathOperator(2, 3);
2.Sum sum3 = (Sum) sum2;
3.System.out.println(sum3.sumAbsolute());
```

En éste ejemplo estamos invocando al método sumAbsolute, pero si recordamos ese método sólo está implementado en la clase Sum, y el objeto sum3 realmente es una instancia de MathOperator, claro vamos a tener el mismo error de casting en tiempo de ejecución, pero como vemos el compilador no es capaz de decirnos que estamos tratando de invocar a un método que no existe en ese objeto, aquí hacemos uso de instanceof para garantizar el casting:

```
view source
print?
01.MathOperator sum2 = new MathOperator(2, 3);
02.if(sum2 instanceof Sum)
03.{
04.    Sum sum3 = (Sum) sum2;
05.    System.out.println(sum3.sumAbsolute());
06.}
07.else
08.{
09.    System.out.println("Object is not a instance of Sum");
10.}
```

El resultado de la ejecución es:

Object is not a instance of Sum

```

39.    }
40.
41.    public double sumAbsolute(){
42.        return Math.abs(operator1) + Math.abs(operator2);
43.    }
44.}
45.
46.
47. public class Multiply extends MathOperator {
48.
49.    protected Multiply(double operator1, double operator2) {
50.        super(operator1, operator2);
51.    }
52.
53.    @Override
54.    public double calculate() {
55.        return operator1*operator2;
56.    }
57.}

```

Empezamos viendo que Sum y Multiply son una subclase de MathOperator y que implementan el método calculate y además Sum tiene un método propio sumAbsolute, con ésta estructura de clases es perfectamente válido para el compilador crear una instancia de Sum de las siguientes maneras:

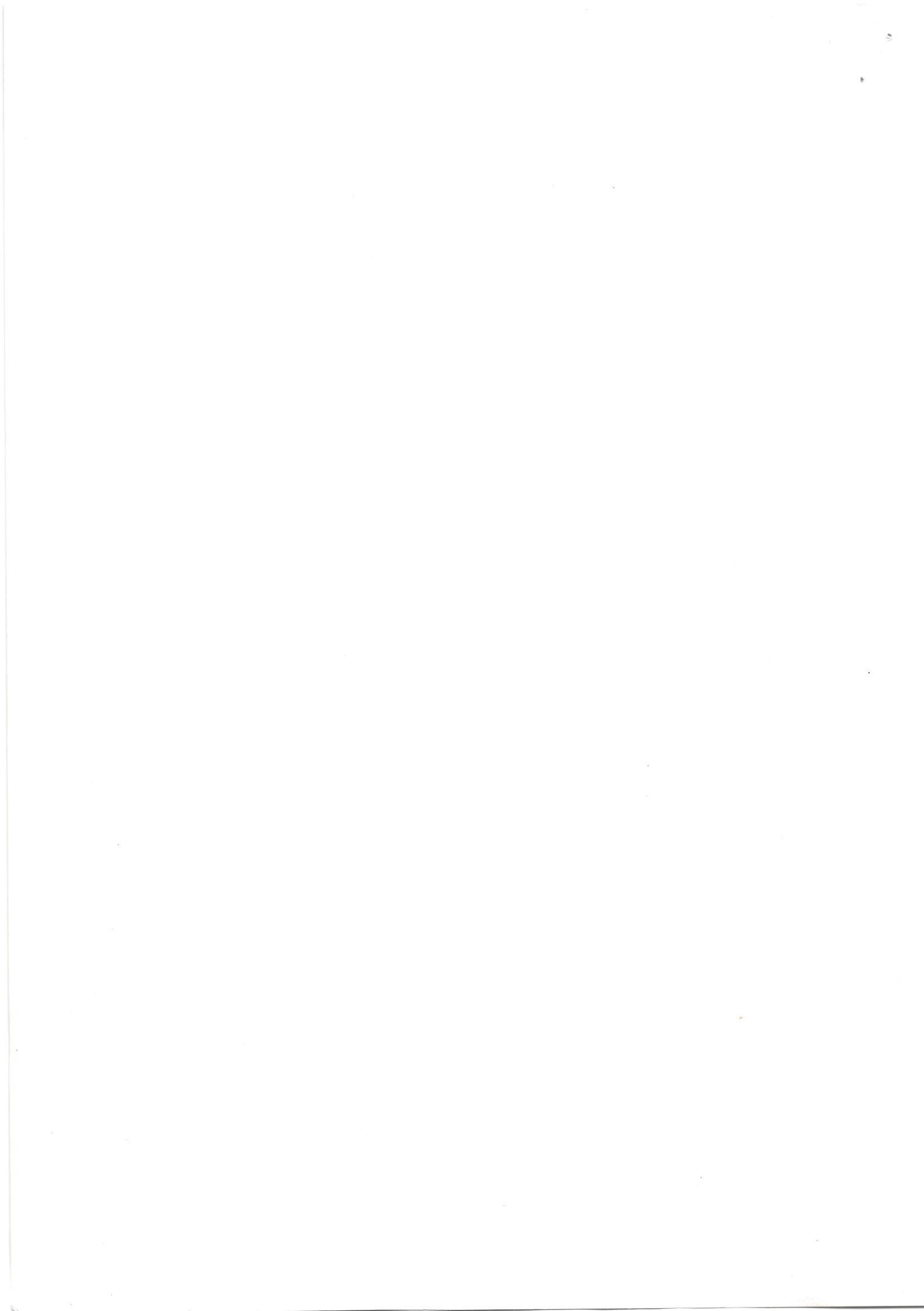
```

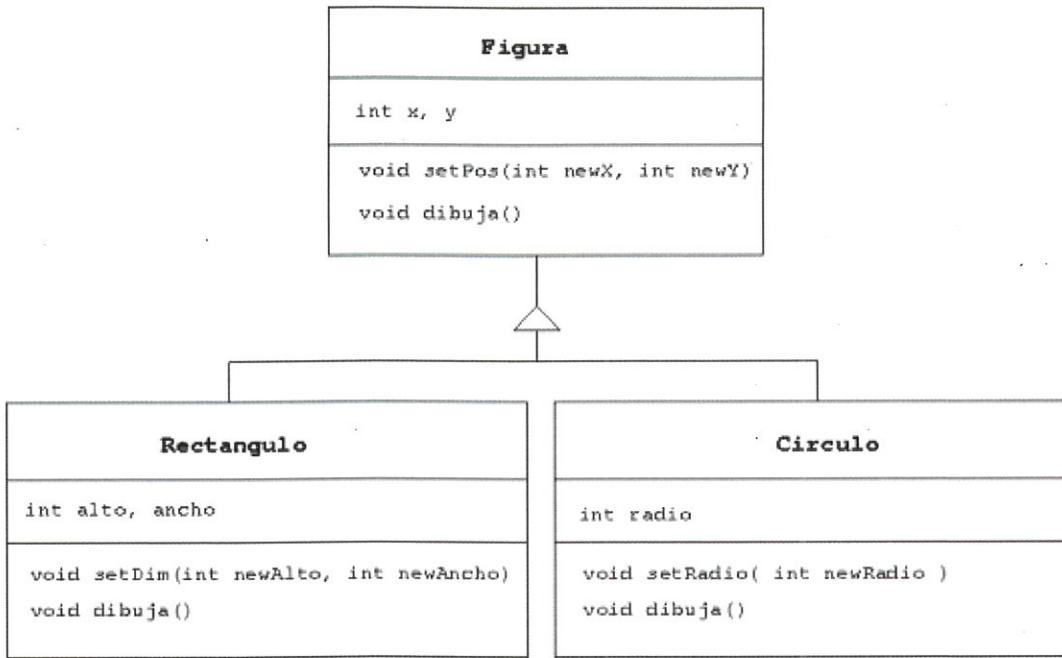
view source
print?
1.Sum sum = new Sum(2, 3);
2.MathOperator sum2 = new Sum(2, 3);
3.Sum sum3 = (Sum) new MathOperator(4,5);

```

En la primera línea es una instancia normal. La segunda línea corresponde a un upcasting, el cual el compilador lo acepta sin problemas inclusive de manera implícita (sin el casting) ya que la superclase MathOperator soporta tener una referencia de cualquier clase que la herede; si recordamos el IS-A, en este caso Sum IS-A MathOperator por lo cual es seguro que cualquier método que se ejecute de MathOperator la clase Sum lo implementa. En la tercera línea tenemos lo contrario, un downcasting, ya que de una superclase estamos tratando de convertirla a un tipo más específico; en este tipo de castings nace una "relación de confianza" entre el desarrollador y el compilador, ya que como vimos en la segunda línea, el objeto sum2 realmente es una instancia de Sum, por eso el compilador acepta el casting como válido.

Si ejecutamos el código el resultado es el siguiente:





En el siguiente ejemplo se muestran distintas conversiones dentro de esta jerarquía, alguna de ellas incorrecta:

```

1 public class ConversionHaciaAbajo {
2
3     public static void main(String[] args) {
4         Figura[] v = new Figura[2];
5         v[0] = new Rectangulo(); // Conversion hacia arriba
6         v[1] = new Circulo(); // Conversion hacia arriba
7         v[0].dibuja(); // Llamada a metodo polimorfico
8         v[1].setRadio(10); // Provoca un error de compilacion.
9
10        // El metodo setRadio no se
11        // encuentra en la clase Figura
12        ((Circulo) v[1]).setRadio(10); // Conversion hacia abajo
13 correcta
14        ((Circulo) v[0]).setRadio(10); // Conversion hacia abajo
15 incorrecta.
16    }
17 }

```

En el ejemplo anterior, para acceder a la interfaz completa del objeto **Circulo** (por ejemplo para acceder al método `setRadio()`) es necesaria la conversión hacia abajo. Ahora bien, hemos de tener la certeza de que el objeto que se encuentra referenciado es realmente del nuevo tipo al que queremos convertirlo, de lo contrario se producirá un error en tiempo de ejecución, tal y como ocurre en la última instrucción.

En este ejemplo es obvio que `v[0]` referencia un **Rectangulo** y `v[1]` un **Circulo**, por lo que es fácil darse cuenta del error cometido en la última instrucción, sin embargo, esto no siempre es así. Hagamos la siguiente modificación sobre el código anterior:

(7.3) casting de clases

CastkingTest.zip

Como ocurre con los tipos básicos, es posible realizar un casting de objetos para convertir entre clases distintas. El uso de esta operación es muy interesante, pero hay que ser muy cuidados al hacerlo y sobre todo comprender perfectamente su funcionamiento.

No se puede convertir objetos de una clase a otra, pero si se utiliza el casting para convertir referencias para indicar la subclase concreta a la que pertenece la misma. Es decir, podemos convertir de objetos de una superclase o objetos de una subclase. Pero sólo si realmente la referencia al objeto pertenece realmente a dicha subclase.

La razón de los casting está en que **es posible asignar referencias de una superclase a objetos de una de sus subclases (pero no al revés)**.

Es complicada esta explicación, por ello es mejor observar este ejemplo:

```
Vehiculo vehiculo5=new Vehiculo();
Coche cocheDePepe = new Coche("BMW");
vehiculo5=cocheDePepe //Esto sí se permite
cocheDePepe=vehiculo5;//ERROR!Tipos incompatibles
cocheDepepe=(Coche)vehiculo5;//Ahora sí se permite, pero sólo
//porque realmente el vehiculo5 hace referencia a un coche
```

Hay que tener en cuenta que los objetos nunca cambian de tipo, se les prepara para su asignación pero no pueden acceder a propiedades o métodos que no les sean propios.

Por ejemplo, si **repostar()** es un método de la clase *coche* y no de *vehículo*:

```
Vehiculo v1=new Vehiculo();
Coche c=new Coche();
v1=c;//No hace falta casting
v1.repostar(5);//iiiError!!! v1 sólo puede utilizar métodos de la
//clase Vehiculo
```

Cuando se fuerza a realizar un casting entre objetos, en caso de que no se pueda realizar, ocurrirá una excepción del tipo **ClassCastException**. Hay que insistir en que realmente sólo se puede hacer un casting si el objeto originalmente era de ese tipo.

Es decir la instrucción:

```
cocheDepepe=(Coche) vehiculo4;
```

Sólo es posible si *vehiculo4* hace referencia a un objeto *Coche*, de otro modo se produce el **ClassCastException**.



Incluso aunque definamos un constructor propio en la clase B:

```
public class B extends A {  
    public B(int v){//ERROR! Falta el constructor A()  
        valor=v;  
    }  
    public void escribir(){  
        System.out.println(valor*2);  
    }  
}
```

¿Por qué sigue ocurriendo el error? Porque el compilador de Java sigue añadiendo de manera implícita la llamada `super()` en los constructores y como no hay constructor por defecto en la clase base, sigue fallando. Para arreglarlo debemos definir constructor por defecto en la superclase o bien realizar una llamada explícita al constructor de la superclase deseado. Ejemplo:

```
public class B extends A {  
    public B(int v){ //Ahora sí!  
        super(v);  
    }  
    public void escribir(){  
        System.out.println(valor*2);  
    }  
}
```

Ya no hay error, ahora el único constructor de `B` invoca explícitamente a un constructor de `A`. En ese caso la invocación sustituye a la llamada `super()`. Ahora ya funciona. Pero, supongamos que la clase B tiene más constructores:

```
public class B extends A {  
    char carácter;  
    public B(int v){  
        super(v);  
    }  
    public B(int v, char c){  
        carácter=c; //ERROR! Falta el constructor A()  
    }  
    public void escribir(){  
        System.out.println(valor*2+" "+carácter);  
    }  
}
```

Otra vez nos metemos en problemas, como ese constructor no tiene invocaciones, Java invoca a `super()` y otra vez no le encuentra.

```
public class Coche extends Vehiculo {  
    public int gasolina;  
    public void acelerar(double kmh) {  
        super.acelerar(kmh)  
        gasolina*=0.9;  
    }  
}
```

En el ejemplo anterior, la llamada `super.acelerar(kmh)` llama al método `acelerar` de la clase `Vehículo` (el cual acelerará la marcha). Es necesario redefinir el método `acelerar` en la clase `coche` ya que aunque la velocidad varía igual que en la superclase, hay que tener en cuenta el consumo de gasolina. Por ello invocamos al método `acelerar` de la clase padre (que no es oculto gracias a `super`).

Se puede incluso llamar a un `constructor` de una superclase, usando la sentencia `super`. Ejemplo:

```
public class Vehiculo{  
    double velocidad;  
    public Vehiculo(double v){  
        velocidad=v;  
    }  
}  
  
public class Coche extends Vehiculo{  
    double gasolina;  
    public Coche(double v, double g){  
        super(v); //Llama al constructor de la clase vehiculo  
        gasolina=g;  
    }  
}
```

(7.2.4) constructores

Los constructores no se heredan de la clase base a las clases derivadas. Pero sí se puede invocar al constructor de la clase base.

De hecho por defecto aunque haya o no haya constructor se hace una invocación al constructor por defecto de la clase base. Ejemplo:

```
public class A {  
    protected int valor;  
    public A(){  
        valor=2;  
    }  
    public void escribir(){  
        System.out.println(valor);  
    }  
}
```

Ejemplo:

```
public class Vehiculo {  
    public int velocidad;  
    public int ruedas;  
    public void parar() {  
        velocidad = 0;  
    }  
    public void acelerar(int kmh) {  
        velocidad += kmh;  
    }  
}  
  
public class Coche extends Vehiculo {  
    public int ruedas=4;  
    public int gasolina;  
    public void repostar(int litros) {  
        gasolina+=litros;  
    }  
}  
  
public class PruebaCoche {  
    public static void main(String[] args) {  
        Coche coche1=new Coche();  
        coche1.acelerar(80); //Método heredado  
        coche1.repostar(12); //Método nuevo, no heredado  
    }  
}
```

(7.2.3) anulación de métodos:

Como se ha visto, las subclases heredan los métodos de las superclases. Pero es más, también los pueden sobrecargar para proporcionar una versión de un determinado método adaptado a las necesidades de la nueva clase.

Por último, si una subclase define un método con el mismo nombre, tipo y argumentos que un método de la superclase, se dice entonces que se **sobrescribe o anula** el método de la superclase.

Si diseñáramos una clase que represente animales en generales, podría tener como métodos: *respirar*, *reproducirse* o *crecer* por ejemplo. Si después quisiéramos una clase para representar leopardos, tendrían métodos como *correr* o *rugir*. Por ello lo que hay que hacer no es definir de nuevo esos métodos, sino simplemente indicar que el leopardo **es un** animal.

Sin herencia la programación orientada a objetos no sería tal, incumpliría uno de sus pilares básicos. La herencia facilita enormemente el trabajo del programador o programadora porque permite crear clases estándar y a partir de ellas crear nuestras propias clases personales. Esto es más cómodo que tener que crear todas las clases desde cero. Además simplifica la reutilización del código y la creación de métodos y elementos más capaces e independientes.

Para que una clase herede las características de otra hay que utilizar la palabra clave **extends** tras el nombre de la clase. A esta palabra le sigue el nombre de la clase cuyas características se heredarán. **En Java sólo se puede tener herencia de una clase.**

Ejemplo:

```
public class Coche extends Vehiculo {  
    ...  
} //La clase Coche, forma parte de la definición de vehículo
```

Las relaciones de herencia se representan así en un diagrama UML (en formato corto):

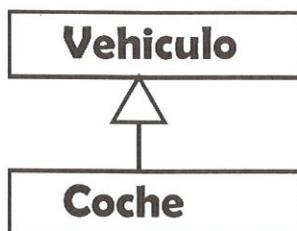


Ilustración 7-6, Diagrama UML de herencia mostrando las clases en formato simple

El diagrama representa que un *Coche* es un *Vehiculo*. Con lo cual los coches tendrán todas las características de los vehículos y además añadirán características particulares.

Puede haber varios niveles de herencia, es decir clases que heredan de otra clase que a su vez es heredera de otra.

Con una cardinalidad fija pero mayor de uno (por ejemplo un usuario siempre alquila 3 libros):

```
public class Usuario{  
    ...  
    Libro libro[]; //representa la relación Usuario->Libro con  
    //cardinalidad 0..1 o 1  
    ...
```

Si la cardinalidad es de tamaño indefinido (como 1..* por ejemplo) entonces la propiedad será una colección de libros (en temas posteriores se habla sobre colecciones de datos).

(7.1.2) agregación y composición

Son asociaciones pero que indican más información que una asociación normal. Definen asociaciones del tipo **es parte de** o **se compone de**.

agregación

Indica que un elemento es parte de otro. Indica una relación en definitiva de composición. Así la clase **Curso** tendría una relación de composición con la clase **Módulo**.



Ilustración 7-4, Diagrama UML de agregación

Cada curso se compone de tres o más módulos. Cada módulo se relaciona con uno o más cursos.

En Java al final se resuelven como las asociaciones normales, pero el diagrama representa esta connotación importante.

composición

La composición indica una agregación fuerte, de hecho significa que una clase consta de objetos de otra clase para funcionar. La diferencia es que cada objeto que compone el objeto grande no puede ser parte de otro objeto, es decir pertenece de forma única a uno.

La existencia del objeto al otro lado del diamante está supeditada al objeto principal y esa es la diferencia con la agregación.

Ejemplo:



Ilustración 7-5, Diagrama UML de composición

(7.8) clases internas	30
(7.8.1) uso de clases internas	30
(7.8.2) clases internas regulares	31
(7.8.3) acceso a propiedades	32
(7.8.4) clases internas a un método	33
(7.8.5) clases internas dentro de un bloque	34
(7.8.6) clases internas anónimas	35
(7.8.7) clases internas estáticas	37
(7.9) creación de paquetes	38
(7.9.1) organización de los paquetes	39

(7.1) relaciones entre clases

Hasta lo visto en los temas anteriores, se puede entender que el diseño de una aplicación es prácticamente el diseño de una clase. Sin embargo en realidad una aplicación es un conjunto de objetos que se relacionan. Por ello en el diagrama de clases se deben indicar la relación que hay entre las clases. En este sentido el diagrama de clases UML nos ofrece distintas posibilidades.

(7.1.1) asociaciones

Las asociaciones son relaciones entre clases. Es decir, marcan una comunicación o colaboración entre clases. Dos clases tienen una asociación si:

- ◆ Un objeto de una clase envía un mensaje a un objeto de la otra clase. Enviar un mensaje, como ya se comentó en el tema anterior es utilizar alguno de sus métodos o propiedades para que el objeto realice una determinada labor.
- ◆ Un objeto de una clase, crea un objeto de otra clase.
- ◆ Una clase tiene propiedades cuyos valores son objetos o colecciones de objetos de otra clase
- ◆ Un objeto de una clase recibe como parámetros de un método objetos de otra clase.

En UML las asociaciones se representan con una línea entre las dos clases relacionadas, encima de la cual se indica el nombre de la asociación y una flecha para indicar el sentido de la asociación. Ejemplo:



Ilustración 7-1, Asociación simple en UML

Como se observa en el ejemplo la dirección de la flecha es la que indica que es el usuario el que alquila los libros. Los números indican que cada usuario puede alquilar de cero a más (el asterisco significa muchos) libros. Esos números se denominan cardinalidad, e indican con cuántos objetos de la clase



