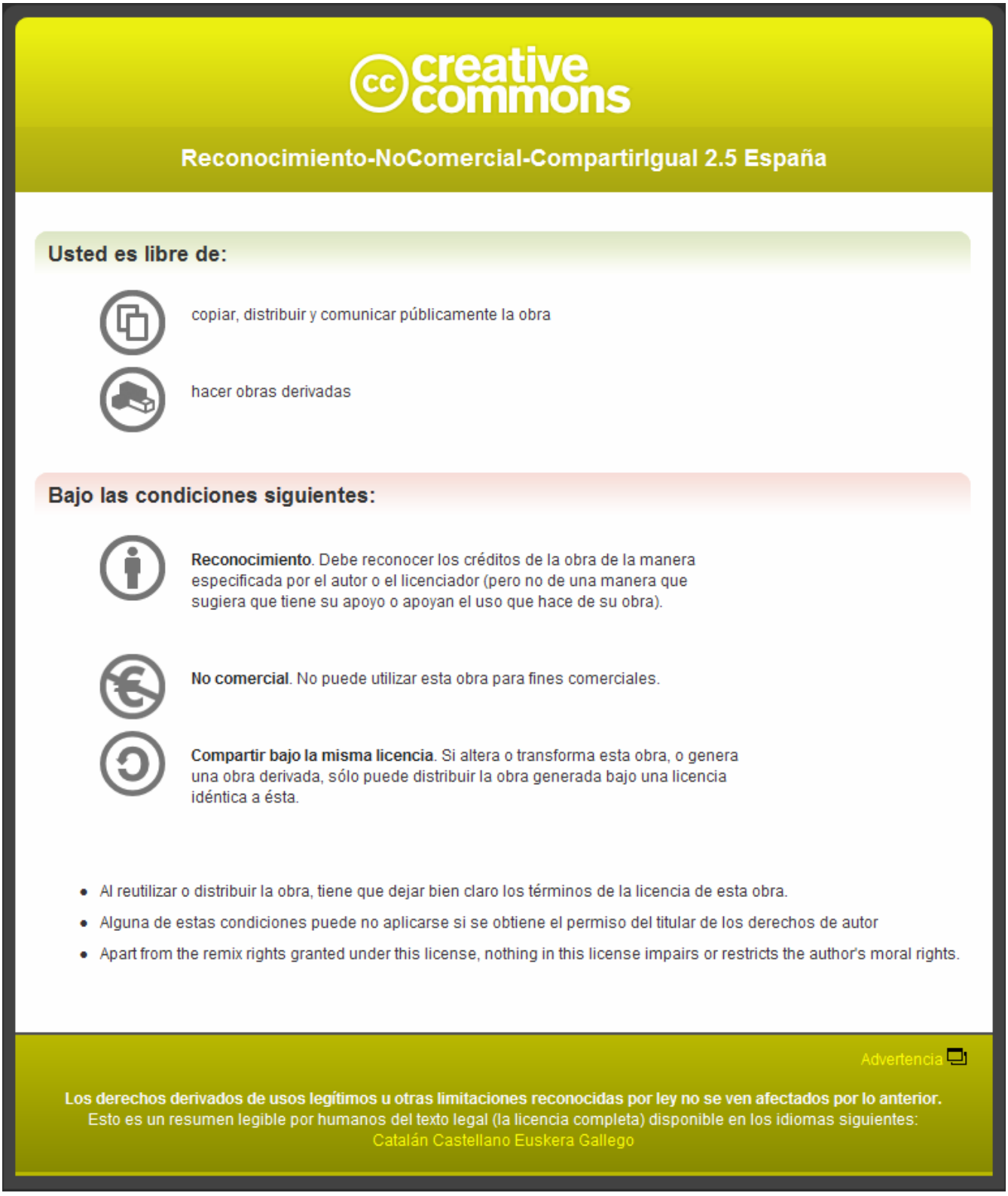
Unidad 7: Herencia y Relaciones entre clases

Fundamentos de Programación. 1º de ASI

Autor: Jorge Sánchez Asenjo (año 2010) http://www.jorgesanchez.net
e-mail:info@jorgesanchez.net



Para ver una copia completa de la licencia, acudir a la dirección http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

(7) herencia y relaciones entre clases

esquema de la unidad

(7.1) relaciones entre clases 6
(7.1.1) asociaciones 6
(7.1.2) agregación y composición 8
(7.2) herencia 9
(7.2.1) introducción 9
(7.2.2) métodos y propiedades heredables. modificador protected 11
(7.2.3) anulación de métodos 12
(7.2.4) constructores 14
(7.3) *casting* de clases 18
(7.3.1) instanceof 19
(7.4) clases abstractas 19
(7.5) final 21
(7.6) interfaces 23
(7.6.1) utilizar interfaces 23
(7.6.2) creación de interfaces 23
(7.6.3) subinterfaces 24
(7.6.4) variables de interfaz 25
(7.6.5) interfaces como funciones de retroinvocación 25
(7.7) la clase Object 26
(7.7.1) comparar objetos. método equals 27
(7.7.2) código hash 28
(7.7.3) clonar objetos 28
(7.7.4) método toString 29
(7.7.5) lista completa de métodos de la clase Object 30

(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:

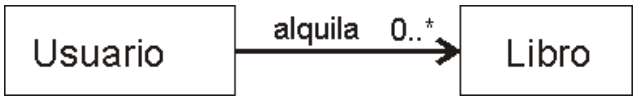


Ilustración 7-1, Asociación simple en UML

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

se puede relacionar cada objeto de la clase que está en la base de la flecha. Puede ser:

- **0..1**. Significa que se relaciona con uno o ningún objeto de la otra clase.
- **0..***. Se relaciona con cero, uno o más objetos
- **1..*** Se relaciona al menos con uno, pero se puede relacionar con más
- ***un número concreto.*** Se puede indicar un número concreto (como 3 por ejemplo) para indicar que se relaciona exactamente con ese número de objetos, ni menos, ni más.

La dirección de la flecha determina de dónde a dónde nos referimos. Así esa misma relación al revés:

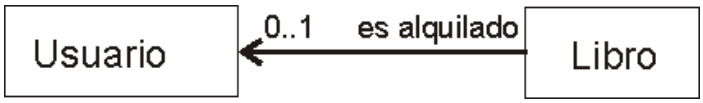


Ilustración 7-2, Asociación UML

Por eso se suele reflejar así de forma completa:

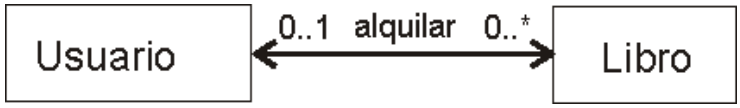


Ilustración 7-3, Asociación UML completa

En muchos casos en las asociaciones no se indica dirección de flecha, se sobreentenderá que la asociación va en las dos direcciones.

Las asociaciones como es lógico implican decisiones en las clases. La clase usuario tendrá una estructura que permita saber qué libros ha alquilado. Y el libro tendrá al menos una propiedad para saber qué usuario le ha alquilado. Además de métodos para relacionar los usuarios y los libros.

Normalmente la solución a la hora de implementar es que las clases incorporen una propiedad que permita relacionar cada objeto con la otra clase. Por ejemplo si la clase usuario Alquila cero o un libro:

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

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:

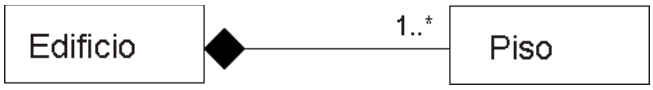


Ilustración 7-5, Diagrama UML de composición

En este caso se refleja que un edificio consta de pisos. De hecho con ello lo que se indica es que un piso sólo puede estar en un edificio. La existencia del piso está ligada a la del edificio. Como se ve la asociación es más fuerte.

implementación

La implementación en Java de clases con relaciones de agregación y composición es similar. Pero hay un matiz importante. Puesto que en la composición, los objetos que se usan para componer el objeto mayor tienen una existencia ligada al mismo, se deben **crear** dentro del objeto grande. Por ejemplo (composición):

```
public class Edificio {
   private Piso piso[];
   public Edificio(.....){
         piso=new Piso[x]; //composición
         ......
```

En la composición (como se observa en el ejemplo), la existencia del piso está ligada al edificio por eso los pisos del edificio se deben de crear **dentro** de la clase ***Edificio*** y así cuando un objeto ***Edificio*** desaparezca, desaparecerán los pisos del mismo.

Eso no debe ocurrir si la relación es de agregación. Por eso en el caso de los cursos y los módulos, como los módulos no tienen esa dependencia de existencia según el diagrama, seguirán existiendo cuando el módulo desaparezca, por eso se deben declarar fuera de la clase cursos. Es decir, no habrá **new** para crear módulos en el constructor. Sería algo parecido a esto:

```
public class Cursos {
   private Módulo módulos[];
   public Edificio(....., Módulo m[]){
         módulos=m; //agregación
         ......
```

(7.2) herencia

(7.2.1) introducción

La herencia define una relación entre clases en la cual una clase posee características (métodos y propiedades) que proceden de otra. Esto permite estructura de forma muy atractiva los programas y reutilizar código de forma más eficiente. Es decir genera relaciones entre clases del tipo **es como**... (también se las llama **es un** en inglés ***is a***).

A la clase que posee las características a heredar se la llama **superclase** y la clase que las hereda se llama **subclase**. Una subclase puede incluso ser superclase en otra relación de herencia.

También se emplea habitualmente los términos **madre** para referirnos a una superclase e **hija** para una subclase (e incluso **abuela** y **nieta**).

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 o 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):

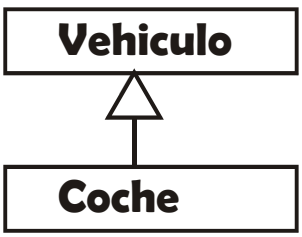


Ilustración 7-6, Diagrama UML de herencia mostrando las clases en formato simple

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.

Ejemplo de herencia múltiple:

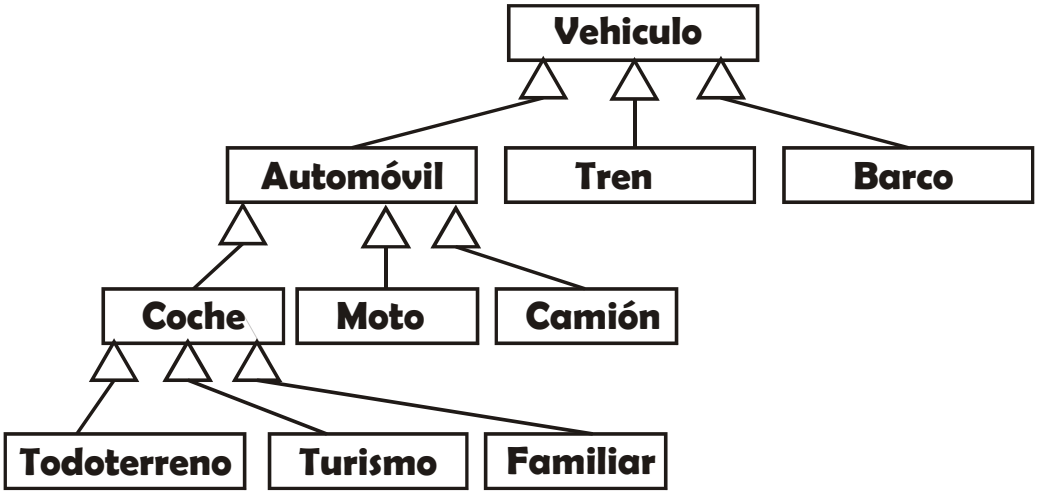


Ilustración 7-7, Diagrama UML de herencia compleja

En ese esquema se puede apreciar como los todoterrenos son coches, que a su vez son automóviles y a su vez son vehículos. Con lo que los todoterrenos adquieren las características heredadas de todas las clases previas.

En Java cada clase sólo puede heredar de una superclase. Eso significa que sólo hay una clase padre. Este tema fue controvertido ya que lenguajes como C++ admiten **herencia múltiple** que significa que una clase puede tener varias clases padre de las que heredaría sus características.

La decisión de no permitir este tipo de herencia se debe a que a la hora de determinar la clase de un objeto habría problemas ya que un objeto se podría considerar perteneciente a varias clases al instanciar.

Las interfaces permiten utilizar cuestiones relativas a la herencia múltiple como se verá más adelante.

(7.2.2) métodos y propiedades heredables. modificador protected

Además de **public**, **private** y ***friendly*** en Java se dispone del modificador **protected** (protegido). Este modificador de acceso está específicamente pensado para la herencia. Cuando se utiliza sobre una propiedad o un método indica que dicha propiedad o método serán visibles por las subclases que además heredarán la propiedad o el método. Sin embargo permanecerán invisibles para el resto.

En la práctica se utiliza más que el privado, ya que las conveniencia de colocar en modo privado las propiedades, las impediría ser heredadas por los descendientes de clase; por ello se utiliza más el protected, para asegurar la descendencia y además mantener la ocultación.

Se heredan todos los métodos y propiedades **protected** y **public** (no se heredan los **private** ni los **friendly**). La subclase por su parte puede definir nuevos métodos y propiedades (es lo habitual).

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.

Ejemplo:

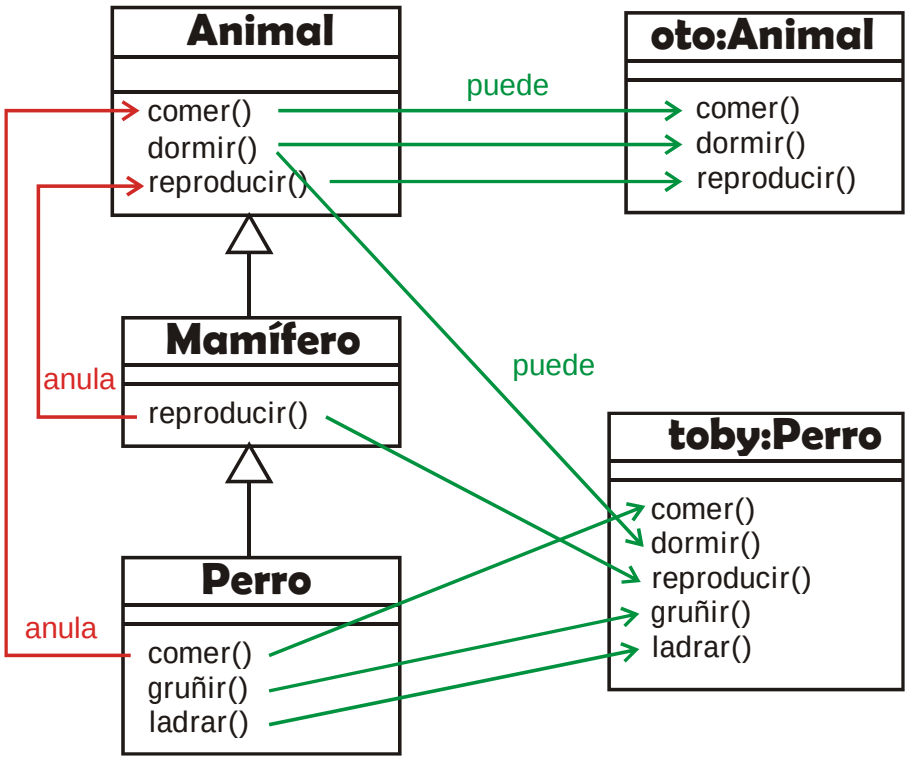


Ilustración 7-8, Anulación de métodos

En el diagrama el objeto ***toby*** que es un perro, usa el método ***comer*** de la clase ***Perro*** (los mamíferos y animales también puede comer pero ***toby*** usa el método redefinido en la clase ***Perro***), sin embargo ***toby*** duerme como cualquier otro animal ya que dormir no ha sido redefinido, se reproduce como un mamífero ya que la reproducción es distinta en un mamífero según el diagrama. Lógicamente puede gruñir y ladrar ya que es un perro.

oto es un animal y por ello todo lo que puede usar procede sólo de la clase **Animal**.

A veces se requiere llamar a un método de la superclase. Eso se realiza con la palabra reservada **super**. Anteriormente hemos visto que **this** hace referencia a la clase actual, bien pues **super** hace referencia a la superclase respecto a la clase actual. con lo que es un método imprescindible para poder acceder a métodos anulados por herencia. Ejemplo:

```
public class Vehiculo {
   public double velocidad;
   ...
   public void acelerar(double kmh) {
          velocidad += kmh;
   }
}
```

```
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);
```

```
    }
}
```

La clase ***A*** simplemente tiene una propiedad (heredable) que se coloca con el valor ***2*** en el constructor por defecto.

```
public class B extends A {
    public void escribir(){
            System.out.println(valor*2);
    }
}
```

La clase ***B*** es derivada de ***A***. Ha redefinido el método escribir y además no tiene constructor. Pero puede utilizar la propiedad ***valor*** que procede de la clase A. Así este código:

```
public static void main(String[] args) {
    A objA=new A();
    B objB=new B();
    objA.escribir(); //escribe 2
    objB.escribir(); //escribe 4
}
```

El método escribir del ***objB*** escribe un 4. La razón, aunque la clase B no tiene constructor, el compilador crea un constructor por defecto cuya única línea será una llamada al constructor de la superclase (***super()***). Ahí es donde valor se pone a dos y como el método escribe dobla el valor al escribir, escribe un cuatro.

Sin embargo si la clase A no tuviera constructor por defecto:

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

```
public class B extends A {//error no se puede invocar al
                           //constructor por defecto
    public void escribir(){
            System.out.println(valor*2);
    }
}
```

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.

Solución:

```
public class B extends A {
    char carácter;
    public B(int v){
            super(v);
    }
    public B(int v, char c){
            this(v); //Ahora sí!! También valdría super(v)
            carácter=c;
    }
    public void escribir(){
            System.out.println(valor*2+" "+carácter);
    }
}
```

Esto puede marear. Pero en realidad las acciones de Java son lógicas. Hay que tener en cuenta permanentemente que en los constructores Java realiza estas acciones:

- Si la primera instrucción de un constructor de una subclase no es una invocación a otro constructor con **this** o **super**, Java añade de forma invisible e implícita una llamada **super()** con la que invoca al constructor por defecto de la superclase. Luego continúa con las instrucciones de inicio de las variables de la subclase y luego sigue con la ejecución normal. Si en la superclase no hay constructor por defecto (sólo hay explícitos) ocurrirá un error.
- Si se invoca a constructores de superclases mediante **super(…)** en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.
- Finalmente, si esa primera instrucción es una invocación a otro constructor de la clase con **this(…)**, entonces se llama al constructor seleccionado por medio de **this** y realiza sus instrucciones, y después continúa con las sentencias del constructor. La inicialización de variables la habrá realizado el constructor al que se llamó mediante **this(..)**.

El uso de **super** y **this** no puede ser simultáneo puesto que ambas tienen que ser la primera instrucción. Lo que significa que hay que elegir entre ambas.

(7.3) casting de clases

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);//¡¡¡Error!!! 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 **ClassCastingException.** 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 **ClassCastingException.**

(7.3.1) instanceof

Permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

```
objeto instanceof clase
```

Comprueba si el objeto pertenece a una determinada clase y devuelve un valor **true** si es así. Ejemplo:

```
Coche miMercedes=new Coche();
if (miMercedes instanceof Coche)
    System.out.println("ES un coche");
if (miMercedes instanceof Vehículo)
    System.out.println("ES un coche");
if (miMercedes instanceof Camión)
    System.out.println("ES un camión");
```

En el ejemplo anterior aparecerá en pantalla:

```
ES un coche
ES un vehiculo
//pero no es un camión
```

(7.4) clases abstractas

A veces resulta que en las superclases se desean incluir métodos teóricos, métodos que no se desea implementar del todo, sino que sencillamente se indican en la clase para que el desarrollador que desee crear una subclase heredada de la clase abstracta, esté obligado a sobrescribir el método.

A las clases que poseen métodos de este tipo (**métodos abstractos**) se las llama **clases abstractas.** Son clases creadas para ser heredadas por nuevas clases creadas por el programador. Son clases base para herencia, plantillas de clases. Las clases abstractas no pueden ser instanciadas (no se pueden crear objetos pertenecientes a clases abstractas).

Una clase abstracta debe ser marcada con la palabra clave **abstract.** Cada método abstracto de la clase, también llevará el **abstract**.

Ejemplo:

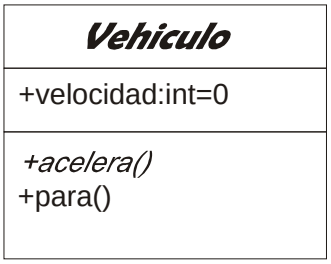
```
public abstract class Vehiculo {
   public int velocidad=0;
   abstract public void acelera();
   public void para() {
         velocidad=0;
   }
}

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 adelantes) 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:



(7.5) modificador final

Ya se ha comentado anteriormente el significado de esta palabra para utilizarse con constantes. No obstante en Java realmente no hay constantes como tal; el significado real de final en un elemento de Java es que dicho elemento no puede cambiar de valor, que es definitivo (en la práctica, una constante).

Pero hay matices dependiendo de donde se utilice final, así:

- **Delante del nombre de una propiedad** o de una variable en su definición, crea una constante. Pero puede ser una constante en **tiempo de compilación** (`final int X=9`) o en **tiempo de ejecución** (`final int X=(int) Math.random()*10+1`). En el último caso el valor de X no se puede cambiar (es constante) pero es un valor que sólo se conocerá en tiempo de ejecución.
- **Delante del nombre de una variable de referencia a objetos**. Es decir se trata de una variable que toma la referencia a un objeto. En este caso también es un valor que se resuelve en tiempo de ejecución que es cuando se crea el objeto. Se trata de instrucciones del tipo:

```
final Coche COCHE=new Coche();
```

En este caso el valor **final** se refiere a que COCHE no puede hacer referencia a otro coche. Sin embargo el valor al que se hace referencia, sí se puede cambiar. Es decir:

```
final Coche COCHE=new Coche();
Coche c=COCHE;
c.mover(200);
```

El coche se ha movido, pese a que la referencia ***COCHE*** es constante. Pero es que el objeto no es constante, es la referencia la que no puede cambiar. ***COCHE*** no puede apuntar a otro objeto, pero el objeto al que apunta sí puede cambiar.

- **Para crear constantes blancas.** Se trata de constantes que se declaran sin indicar el valor:

```
final int X;
```

Que son utilizadas para recoger valores en la construcción y hacer que ese valor sea inmutable:

```
public Constructor(int i) {
   X=i;   //ese valor es el tomado como constante, X no podrá
          //cambiarlo
}
```

- **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 invocarles 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.

(7.6) interfaces

En lenguajes como C++, las clases pueden heredar de varias clases. La limitación de que sólo se puede heredar de una clase, hace que haya problemas ya que muchas veces se deseará heredar de varias clases. Aunque ésta no es la finalidad directa de las interfaces, sí que tiene cierta relación

Mediante interfaces se definen una serie de **comportamientos** de objeto. Estos comportamientos puede ser **implementados** en una determinada clase. No definen el tipo de objeto que es, sino lo que pueden hacer (sus capacidades). Por ello lo normal es que el nombre de las interfaces terminen con el sufijo **able** (***configurable***, ***modificable***, ***cargable***).

Por ejemplo en el caso de la clase ***Coche***, esta deriva de la superclase Vehículo, pero además puesto que es un vehículo a motor, podría implementar métodos de una interfaz llamada por ejemplo ***Arrancable*****.** Se dirá entonces que la clase ***Coche*** es ***arrancable***.

En Java el manejo de interfaces es fundamental para comprender íntegramente todos los elementos del lenguaje. Las interfaces juegan un papel fundamental en la creación de aplicaciones Java. Permiten establecer un protocolo de funcionamiento de una serie de clases y eso facilita su uso y comprensión.

(7.6.1) utilizar interfaces

Para hacer que una clase utilice una interfaz, se añade detrás del nombre de la clase la palabra **implements** seguida del nombre del interfaz. Se pueden poner varios nombres de interfaces separados por comas (solucionando, en cierto modo, el problema de la herencia múltiple).

```
class Coche extends Vehiculo implements Arrancable {
   public void arrancar (){
          ....
   }
   public void detenerMotor(){
          ....
   }
}
```

Hay que tener en cuenta que la interfaz ***Arrancable*** no tiene porque tener ninguna relación de herencia con la clase ***Vehículo***, es más se podría implementar el interfaz ***Arrancable*** a una bomba de agua.

(7.6.2) creación de interfaces

Una interfaz en realidad es una serie de **constantes y métodos abstractos**. Es más el parecido entre interfaces y clases abstractas es muy grande. Cuando una clase implementa un determinado interfaz **debe** definir los métodos abstractos de éste. Esta es la base de una interfaz, en realidad no hay una

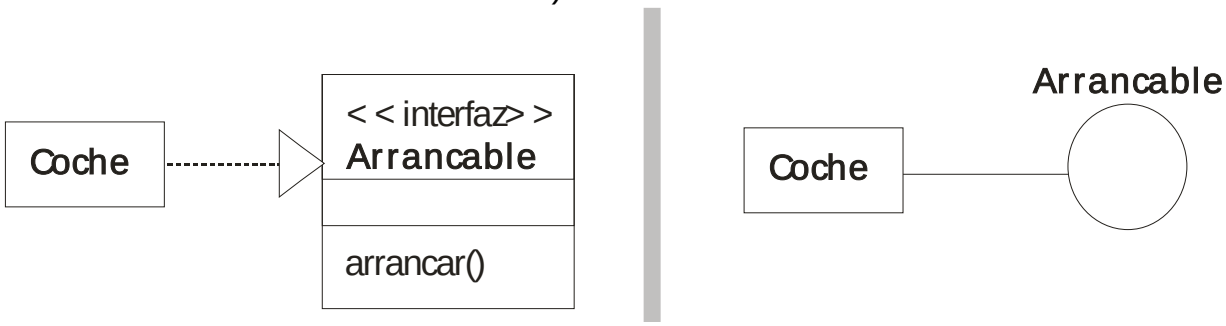
relación sino que hay una **obligación** por parte de la clase que implemente la interfaz de redefinir los métodos de ésta.

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).



Forma completa Forma abreviada

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***.

(7.6.4) variables de interfaz

Las variables de interfaz no son objetos. Se parece su uso, pero en realidad son variables que son capaces de hacer referencia a cualquier objeto de una clase que haya implementado la interfaz. Es decir una variable de la interfaz ***Arrancable*** puede utilizarse para hacer referencia a objetos de la clase ***Coche*** y de la clase ***BombaAgua***, ya que ambas son clases que implementan dicha interfaz.

Esto permite cosas como:

```
Arrancable motorcito; //motorcito es una variable de tipo
                      // arrancable
Coche c=new Coche(); //Objeto de tipo coche
BombaAgua ba=new BombaAgua(); //Objeto de tipo BombaAgua
motorcito=c; //Motorcito apunta al coche
motorcito.arrancar(); //Se arrancará el coche
motorcito=ba; //Motorcito apunta a la bomba de agua
motorcito.arrancar(); //Se arranca la bomba de agua
```

El juego que dan estas variables es impresionante, manipulan todo tipo de objetos. Eso facilita la escritura de métodos genéricos para clases que implementen la misma interfaz

(7.6.5) interfaces como funciones de retroinvocación

En C++ una función de retroinvocación es un puntero que señala a un método o a un objeto. Se usan para controlar eventos. En Java se usan interfaces para este fin. La idea es utilizar los métodos de la interfaz sobre objetos de los que no conocemos la clase (uno de los usos más interesantes de las interfaces). Ejemplo:

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

```
class Texto implements Escribible {
   ...
   public void escribe(String texto){
         System.out.println(texto);
   }
}
```

```
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,....

y para ello habrá que redefinir esos métodos a fin de que se ajusten las necesidades particulares de cada clase.

(7.7.1) comparar objetos. método equals

La clase **Object** proporciona un método para comprobar si dos objetos son iguales. Este método es **equals.** Este método recibe como parámetro un objeto con quien comparar y devuelve **true** si los dos objetos son iguales.

No es lo mismo equals que usar la comparación de igualdad. Ejemplo (suponiendo que se ha definido correctamente el método equals):

```
Coche uno=new Coche("Renault","Megane","P4324K");
Coche dos=uno; //dos y uno son referencias al mismo coche
boolean resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado también valdrá true
dos=new Coche("Renault","Megane","P4324K"); //los mismos datos
resultado=(uno.equals(dos)); //Resultado valdrá true
resultado=(uno==dos); //Resultado ahora valdrá false
```

En el ejemplo anterior **equals** devuelve **true** si los dos coches tienen el mismo modelo, marca y matrícula . El operador "==" devuelve **true** si las dos referencias que se comparan apuntan al mismo objeto.

Realmente en el ejemplo anterior la respuesta del método **equals** sólo será válida si en la clase que se está comparando (*Coche* en el ejemplo) se ha redefinido el método **equals**. Esto no es opcional sino **obligatorio si se quiere usar este método.**

El resultado de equals depende de cuándo consideremos nosotros que devolver verdadero o falso (es el creador de la clase el que sabe como compararla).

Para que el ejemplo anterior funcione, el método equals de la clase Coche sería:

```
public class Coche extends Vehículo{
   ...
   public boolean equals (Object o){
         if ((o!=null) && (o instanceof Coche)){
               if (((Coche)o).matricula==matricula &&
                     ((Coche)o).marca==marca  &&
                     ((Coche)o).modelo==modelo))
                           return true;
               else
                           return false
         }
         else
               return false; //Si no se cumple todo lo anterior
   }
}
```

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 incluída 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;
          }
}
....
```

```
//Clonación
    Coche uno=new Coche();
    Coche dos=(Coche)uno.clone();
```

En la última línea del código anterior, el cast *(Coche)* es obligatorio ya que clone devuelve forzosamente un objeto tipo **Object**.

Esta es una definición vaga del método clone ya que utiliza el clone de la propia clase **Object**. No es lo habitual. Lo habitual es arreglar los problemas que provoca clone con las referencias a objetos internas:

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

Suponiendo que los coches constan de una propiedad llamada *motor* de clase *Motor*, clase que a su vez es *clonable*; el código anterior sería el correcto para clonar, de otro modo con el código anterior a éste, los dos coches, el original y el clonado, compartirían motor.

(7.7.4) método toString

Este es un método de la clase Object que da como resultado un texto que describe al objeto. La utiliza, por ejemplo el método **println** para poder escribir un método por pantalla. Normalmente en cualquier clase habría que definir el método **toString**. Sin redefinirlo el resultado podría ser:

```
Coche uno=new Coche();
System.out.println(uno);//Escribe: Coche@26e431  el código de clase
```

Si redefinimos este método en la clase Coche:

```
public String toString(){
   return("Velocidad :"+velocidad+"\nGasolina: "+gasolina);
}
```

Ahora en el primer ejemplo se escribiría la velocidad y la gasolina del coche.

(7.7.5) lista completa de métodos de la clase Object

método	significado
protected Object **clone()**	Devuelve como resultado una copia del objeto.
boolean **equals(**Object obj**)**	Compara el objeto con un segundo objeto que es pasado como referencia (el objeto ***obj***). Devuelve **true** si son iguales.
protected void **finalize()**	Destructor del objeto
Class **getClass()**	Proporciona la clase del objeto. Permite saber el nombre de la clase
int **hashCode()**	Devuelve un valor ***hashCode*** para el objeto
void **notify()**	Activa un hilo (thread) sencillo en espera.
void **notifyAll()**	Activa todos los hilos en espera.
String **toString()**	Devuelve una cadena de texto que representa al objeto
void **wait()**	Hace que el hilo actual espere hasta la siguiente notificación
void **wait(**long tiempo**)**	Hace que el hilo actual espere hasta la siguiente notificación, o hasta que pase un determinado tiempo
void **wait(**long tiempo, int nanos**)**	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
- Usas clases anónimas para hacer definiciones sobre necesidades en ejecución
- Casi imprescindibles para manejar eventos

(7.8.2) clases internas regulares

Las clases internas regulares son simplemente clases definidas dentro de otras clases (sean de forma pública o privada). Al definir una clase dentro de otra, estamos haciéndola totalmente dependiente. Normalmente se realiza esta práctica para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.

Por ejemplo:

```
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;
    }
  }
}
```

El objeto ***motor*** pertenece a la clase ***Motor*** que es interna a la clase ***Coche***. Si quisiéramos acceder al objeto motor de un coche sería:

```
Coche c=new Coche(1200);
System.out.println(c.motor.cilindrada);//Saldrá 1200
```

Las clases internas pueden ser privadas, protegidas o públicas.

Al compilar un clase que contiene clases internas se crean dos archivos **.class**, en el ejemplo sería ***Coche.class*** y ***Coche$Motor.class***. Sin embargo el comando **java Coche$Motor** generaría un error ya que no podemos acceder a esa clase para su ejecución ya que no se puede definir nada estático en ella (y por lo tanto **main** tampoco).

Los objetos de la clase interna (en inglés **inner class**) sólo tienen sentido dentro de la clase contenedora o externa (en inglés **outer class**). Pero se pueden crear fuera si primero creamos un objeto de la clase contenedora. La sintaxis es compleja, sería así:

```
Coche c=new Coche (150);
Coche.Motor m=c.new Motor (200);
System.out.println(m.cilindrada);//200
System.out.println(c.motor.cilindrada);//150
```

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;
  }
```

```
  public class Motor{
    public int cilindrada;
    public Motor(int cil){
      Coche.this.cilindrada=cil;//Coche
      this.cilindrada=cil;//Motor, también vale sin la palabra this
    }
  }
}
```

(7.8.4) clases internas a un método

Se trata de clases que se definen dentro de un método. Sólo se pueden utilizar dentro del método. Su uso es un tanto controvertido. Ejemplo de uso:

```
public class Externa {
   private String nombre = "Externa";
   int aleatorio(){
         class Interna extends Random{
               Interna(){
                     super();
               }
               public int aleatorioPar(){
                     return nextInt()*2;
               }
         }
         Interna i=new Interna();
         return i.aleatorioPar();
   }
   public static void main(String[] args) {
         Externa e=new Externa();
         System.out.println(e.aleatorio());
   }
}
```

En el código anterior la clase ***Interna*** se crea dentro del método ***aleatorio***. La clase sólo está disponible para el método. Esta clase hereda de la clase **Random** el método **nextInt** (entre otros) que permite obtener un número entero aleatorio.

En el ejemplo esta técnica no parece muy útil ya que es evidente que lo que realiza la función aleatoria, se puede hacer de otras maneras (sin crear una clase aleatoria heredera de la clase **Random**).

La cuestión es ¿por qué utilizar este tipo de clases internas? Normalmente se usan cuando el método intenta solucionar un problema complicado y necesita apoyarse en una clase, pero no se necesita que esta clase esté disponible fuera. Son por tanto clases que quedan fuera del diseño.

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.

Lo único que hace la clase creada es enviar el número 25 al propio ordenador a través del puerto de red 1200. Esto se podría hacer de forma más sencilla, pero se ha considerado hacer de esta forma y es completamente válido.

La clase *Msg* se ha creado dentro del **if** por ello sólo se puede utilizar en el bloque interior al **if**. Por eso intentar hacer un nuevo mensaje fuera de este bloque causa un error.

El uso de este tipo de clases internas es aún más limitado, nuevamente se usan sólo cuando se considera que ayudan a solucionar el problema. Pero es difícil encontrar situaciones reales para su uso.

(7.8.6) clases internas anónimas

Son el caso más curioso de clases internas. Se trata de clases que sirven para crear objetos cuya clase se crea sobre la marcha sin nombre. Un equivalente al ejemplo anterior mediante clase anónima es:

```
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){
      try{
            new Socket("localhost",1200) {
                  public void enviar() throws IOException {
                        super.sendUrgentData(25);
                  }
            }.enviar();
      }
      catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Error");
       }
    }
   }while(i!=2);
}
```

No es el ejemplo ideal de clase anónima, pero sin duda es curioso. En el **if** se crea un objeto que desaparecerá cuando finalice el bloque ya que no se asigna a nadie.

En realidad es un objeto de tipo **Socket**, pero en lugar de simplemente crearle y utilizar después los métodos de esa clase, resulta que hay una llave justo detrás de la invocación al constructor.

Esa llave (en el código marcado en naranja fuerte) inicia la definición de la clase anónima. En esa definición se crea el método enviar para enviar el número 25 a través de la red.

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);
 }
}
```

La definición de las clases anónimas está resaltado en naranja. Las referencias ***msg*** y ***msg2*** son de tipo ***Escribible***, pero cada uno tiene una implementación distinta a través de la clase anónima. El primero permite escribir en la consola y el segundo en una ventana de mensaje.

Es muy importante tener en cuenta que el cierre de las llaves tiene que incorporar el punto y coma, ya que son clases definidas dentro de una instrucción de creación de objeto que tiene que finalizar como lo hace normalmente, es decir con el punto y coma.

Este código es muy interesante ya que permite observar el funcionamiento de estas clases. En principio la creación de los objetos parece normal hasta que llega la llave, entonces todo el código cambia y lo que parecía una instrucción sencilla de creación de objetos, se convierte en un código de creación de clase con una nueva implementación de la interfaz ***Escribible***.

Esta definición de clase sólo está disponible en cada objeto que se ha creado, por lo que su uso parece muy limitado, de hecho lo es. En la práctica real implementar de forma muy particular una interfaz, una clase o una clase abstracta suele ser el uso más habitual de este tipo de clases.

(7.8.7) clases internas estáticas

En realidad no son clases internas. Se las llama también clases anidadas estáticas para evitar el uso de la palabra ***interna***, pero en general todo el mundo las conoce como clases internas estáticas.

Estas clases no mantienen ningún vínculo con la clase contenedora. Se pueden crear objetos de la clase interna sin crear ningún objeto contenedor. Ejemplo:

```
public class Coche {
  public int velocidad;
  public Motor motor;
  public Coche(int cil) {
    motor=new Motor(cil);
    velocidad=0;
  }
  public static class Motor{ //Clase interna
    public int cilindrada;
    public Motor(int cil){
      cilindrada=cil;
    }
  }
}
public class PruebaCoche{
   public static void main(String[] args) {
         Coche.Motor m=new Coche.Motor(2000);
         System.out.println(m.cilindrada);
   }
}
```

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;
```

Gracias a esta instrucción para utilizar la clase Coche no hace falta indicar el paquete en el que se encuentra, basta indicar sólo ***Coche***. Se puede utilizar el símbolo asterisco como comodín.

Ejemplo:

```
import locomoción.*;
//Importa todas las clase del paquete locomoción
```

Esta instrucción no importa el contenido de los paquetes interiores a ***locomoción*** (es decir que si la clase ***Coche*** está dentro del paquete *motor*, no sería importada con esa instrucción, ya que el paquete motor no ha sido importado, sí lo sería la clase ***locomoción.BarcoDeVela***). Por ello en el ejemplo lo completo sería:

```
import locomoción.*;
import locomoción.motor.*;
```

Cuando desde un programa se hace referencia a una determinada clase se busca ésta en el paquete en el que está colocada la clase y, si no se encuentra, en los paquetes que se han importado al programa. Si ese nombre de clase se ha definido en un solo paquete, se usa. Si no es así podría haber **ambigüedad** por ello se debe usar un prefijo delante de la clase con el nombre del paquete.

Es decir:

```
paquete.clase
```

O incluso:

```
paquete1.paquete2.....clase
```

En el caso de que el paquete sea subpaquete de otro más grande.

En el caso de los nombres de paquete, todo el nombre debe estar en minúsculas aunque lo formen varias palabras (por ejemplo ***mipaquete***).

(7.9.1) organización de los paquetes

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello es necesario usar la variable de entorno **CLASSPATH**. Esta variable se debe definir en los archivos de inicio del sistema (como se comentó en el primer tema, véase **(1.3.3) instalación del SDK**., Hay que añadirla las rutas a las carpetas que contienen los paquetes (normalmente todos los paquetes se suelen crear en la misma carpeta), a estas carpetas se las suele llamar **filesystems** de Java, nomenclatura muy habitual en los sistemas Linux/Unix. Ejemplo de classpath (Windows):

```
CLASSPATH=.;C:\Users\Jorge\java;D:\paquetesJava
```

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 almacena 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 .

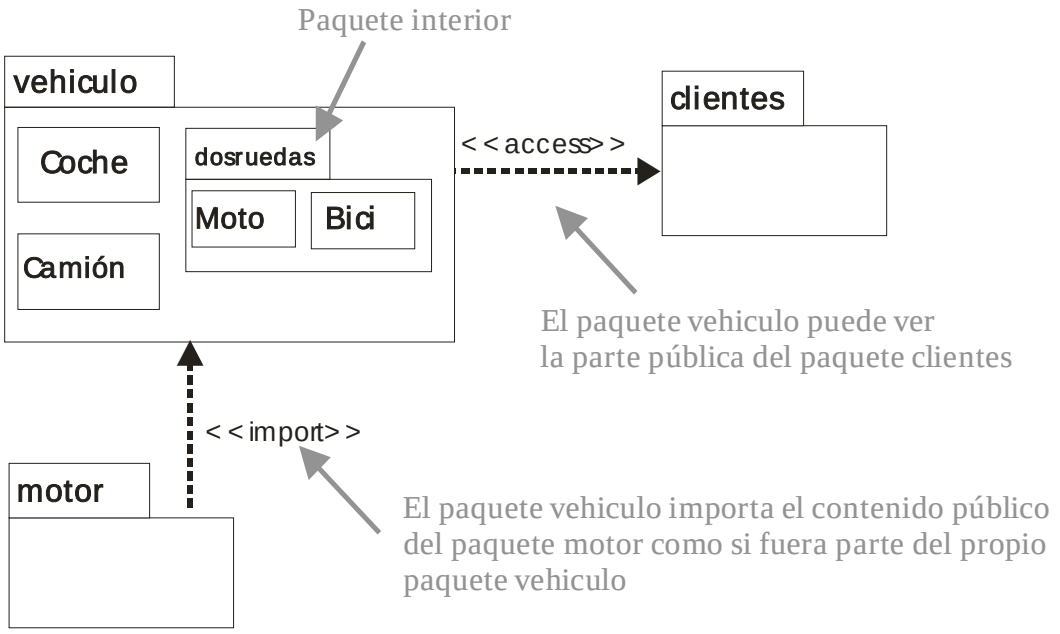


Ilustración 7-10, diagrama de paquetes UML