Programación Avanzada
UD5

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

01 de Octubre de 2012

Copyright ©2012- P. Pablo Garrido Abenza

ÍNDICE GENERAL

Copyright ©2012- P. Pablo Garrido Abenza

CAPÍTULO 5

PROGRAMACIÓN ORIENTADA A OBJETOS (II)

Copyright ©2012- P. Pablo Garrido Abenza

En un tema anterior ya se presentaron los conceptos básicos de la Programación Orientada a Objetos (POO). En este capítulo se presentan conceptos más avanzados de la POO, como son el uso paquetes para la organización de clases (`package`), herencia (`extends`), clases abstractas (`abstract`), e interfaces (`interface`). Finalmente, se verán las enumeraciones (`enum`).

5.1. Paquetes

En este apartado se explica cómo empaquetar una serie de clases en paquetes y cómo utilizarlos. Con objeto de facilitar la reutilización de código es necesario tener algún sistema para clasificar un conjunto de clases (e interfaces) según su funcionalidad. El mecanismo que ofrece Java es el uso de paquetes (`package`). Un paquete puede contener una serie de clases e interfaces, así como otros paquetes, de forma recursiva, formando así una estructura jerárquica. El paquete raíz de esta jerarquía de paquetes es el paquete llamado `java`, el cual no incorporpora ninguna clase pero sí el resto de paquetes. En capítulos anteriores, habíamos necesitado utilizar algún paquete, como por ejemplo el `java.util`, haciendo referencia al paquete `util` que se incluye dentro del paquete raiz `java`.

Alguna ventaja del uso de paquetes:

- Es más fácil de encontrar una determinada clase, puesto que estarán agrupadas en paquetes según su funcionalidad.
- Las clases que pertenecen a un mismo paquete pueden acceder libremente a los miembros de otras clases del mismo paquete, en el caso de no utilizar ningún modificador para restringir el acceso a sus miembros (acceso a nivel paquete o acceso amigable).
- Los nombres de las clases dentro de un paquete no entrarán en conflicto con nombres ya existentes en otros paquetes, esto es, pueden existir otras clases con el mismo nombre dentro de otros paquetes. Esto es así porque cada paquete crea un nuevo espacio propio para los nombres de las clases.

Las **clases de Java** están agrupadas en paquetes, como por ejemplo:

- `java.lang`: paquete del lenguaje Java (*Java Language Package*). Contiene clases básicas del lenguaje, como por ejemplo, dos clases que han sido utilizadas anteriormente como la clase `System` y la clase `Math`. El compilador importa automáticamente este paquete en todos los programas, por lo que no es necesario hacerlo manualmente; por ejemplo, para utilizar las clases anteriormente mencionadas no habíamos tenido que hacer nada en especial.
- `java.util`: paquete de utilidades de Java (*Java Utilities Package*). Contiene ciertas clases para manejar estructuras de datos (listas, pilas, colas, etc.) y otras clases generales que pueden ser de utilidad, como la clase `Arrays` o la `StringTokenizer` utilizadas para manipular arrays y `String`, respectivamente. Para utilizar las clases de este y el resto de paquetes es necesario importarlas, tal como vimos en algún ejemplo y se explica a continuación.
- `java.io`: paquete de entrada/salida (input/output) de Java (*Java Input/Output Package*). Contiene clases de acceso a ficheros (archivos y flujos).
- `java.net`: paquete de trabajo con redes de Java (*Java Networking Package*). Contiene clases basadas en el protocolo TCP/IP para la comunicación de sistemas, por ejemplo *sockets* UDP y TCP.
- `java.awt`: paquete de clases AWT (*Abstract Window Toolkit Package*). Contiene una serie de clases para desarrollar aplicaciones gráficas en varias plataformas: Windows, MacOS X, Motif (Unix).
- `java.applet`: paquete de *applets* de Java (*Java Applet Package*). Necesario para realizar *applets* ejecutables por los navegadores de Internet.

Las **clases realizadas por nosotros** también pueden agruparse en paquetes; en caso de no especificar nombre de paquete (`package`) en una clase, ésta pertenecerá al paquete predeterminado o sin nombre (*default*); en el IDE NetBeans aparece indicado como <paquete predeterminado>. En la siguiente sección se explicará el concepto de herencia, y veremos que las clases también forman una jerarquía, pero son dos jerarquías independientes, no tiene nada que ver una con la otra.

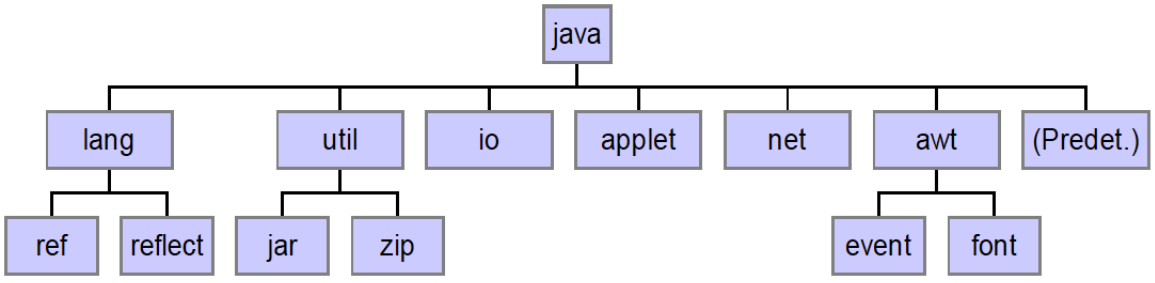


Figura 5.1: Jerarquía de paquetes de Java - Paquete raíz `java.lang`

Copyright © 2012 P. Pablo Garrido Abenza

Declaración y uso de paquetes

Para especificar que una clase pertenece a un paquete, debemos utilizar una sentencia `package` como primera línea de código del fichero (sin tener en cuenta líneas de comentarios). Por ejemplo, para indicar que una clase pertenece al paquete `programacionavanzada` pondríamos:

```
1 package programacionavanzada;
2
3 class MiClase {
4     ...
5 }
```

Como se acaba de comentar, no es obligatorio especificar que una clase pertenece a un paquete mediante la sentencia `package`, pero en ese caso la clase pertenecerá al paquete predeterminado.

Para que una clase pueda utilizar un clase determinada de un paquete es necesario importarla mediante una sentencia `import`, pudiendo utilizar tantas sentencias `import` como sean necesarias. Estas sentencias se escriben después de la sentencia `package` pero antes de la clase; la sentencia `package` y las sentencias `import` es el único código que va fuera de la clase. En caso de requerir importar varias clases de un mismo paquete, en vez de importarlas una a una podemos importarlas todas utilizando un asterisco `'*'` como nombre de la clase. En los siguientes ejemplos importamos dos clases del paquete `java.util`, primero uno a uno (a la izquierda), y luego todas las clases del paquete que sean necesarias (a la derecha). Utilizar el asterisco `'*'` no quiere decir que nuestro programa final vaya a ser de mayor tamaño que importando las clases concretas que se vayan a utilizar; simplemente es una forma más cómoda al tener que escribir menos, ya que el compilador únicamente importará finalmente aquellas clases que se han utilizado en la clase, requiriendo quizás algo más de tiempo a la hora de compilar la clase.

```
1 package programacionavanzada;
2
3 import java.util.Arrays;
4 import java.util.StringTokenizer;
5
6 class MiClase {
7     ...
8 }
```

```
1 package programacionavanzada;
2
3 import java.util.*;
4
5 class MiClase {
6     ...
7 }
```

Cuando se importa un paquete con `'*'` tan sólo se importan las clases, no otros paquetes que a su vez pudiera contener. Por ejemplo, si se importa el paquete `java.util.*`, y se necesitase también el paquete `java.util.zip.*`, habría que importarlo de forma separada.

```
1 import java.util.*;
2 import java.util.zip.*;
```

Copyright © 2012 P. Pablo Garrido Abenza

NOTA: por conveniencia, las clases del paquete `java.lang` se importan automáticamente, no es necesario utilizar la sentencia `import` para poder utilizarlas, aunque sería válido. Por ejemplo, viendo el primer ejemplo `HolaMundo`, se utilizaba la clase `System` para poder invocar al método `println()`, y no fue necesario importar dicha clase.

Alternativamente, sería posible utilizar una clase sin importarla, pero con la condición de **referenciar completamente su nombre**, es decir, especificando toda la jerarquía de paquetes separados con un punto y el nombre de la clase cada vez que se vaya a utilizar cualquiera de sus métodos. Por ejemplo, si utilizásemos el método `sort()` de la clase `Arrays` para ordenar un array, quizás sería más conveniente utilizar su nombre totalmente referenciado en vez de importar la clase del paquete `java.util` del siguiente modo:

```
1  // Importando la clase Arrays
2  import java.util.Arrays;
3
4  class MiClase {
5     public static void main (String
          args[]) {
6        int[] n = {4, 1, 5, 7};
7        ...
8        Arrays.sort (n);
9        ...
10    }
11 }
```

```
1  // Sin importar la clase Arrays
2  // (referenciandola completamente)
3
4  class MiClase {
5     public static void main (String
          args[]) {
6        int[] n = {4, 1, 5, 7};
7        ...
8        java.util.Arrays.sort (n);
9        ...
10    }
11 }
```

Una situación en la que sería obligatorio referenciar completamente el nombre de una clase sería en caso de conflicto de nombres entre dos clases que se llamen igual de distintos paquetes. Por ejemplo, el paquete `java.util` incorpora una clase llamada `List`, para gestionar listas de objetos dinámicas; por otro lado, la clase `java.awt` incorpora una clase llamada también `List`, que representa un objeto gráfico lista desplazable. El compilador detectará tal ambigüedad, requiriendo referenciar completamente la clase `List` que se desea utilizar en cada momento, aunque bastaría con referenciar completamente la última que se haya importado.

```
1  import java.util.*;
2  import java.awt.*;
3
4  class MiClase {
5     public static void main (String args[]) {
6        List o1 = new List();   // Se supone que es java.util.List
7        java.awt.List o2 = new java.awt.List();
8        ...
9     }
10 }
```

Copyright © 2012 P. Pablo Garrido Abenza

Recomendación para nombrar paquetes

Sería muy probable que dos programadores utilizasen el mismo nombre para una clase similar, pero no habría problema si el paquete al que pertenecen fuese diferente. Puesto que en Java es muy fácil que se utilicen clases desarrolladas por otros programadores gracias al uso de los llamados `applets`, podríamos encontrarnos con algún conflicto en tal caso. Para asegurarnos de que cada programador utiliza un nombre diferente para un paquete se aconseja seguir una serie de recomendaciones.

Para empezar, por convenio, los nombres de los paquetes se escriben totalmente en minúsculas, para evitar conflicto con el nombre de las clases, las cuales, por convenio, comienzan escribiéndose en mayúscula y el resto en minúscula. Además, no deberían comenzar por `java.` o `javax.`, ya que es el comienzo siempre de los paquetes de la jerarquía de paquetes de Java y sus extensiones.

Se recomienda utilizar nuestro dominio de Internet en orden inverso, seguido de la jerarquía del paquete en sentido descendente. De esta forma se garantiza que el nombre del paquete será único, puesto que las direcciones URL son únicas. Además, esto permitirá conocer quien fue el desarrollador. Por ejemplo, si algún programador de la UMH (http://www.umh.es) hubiera desarrollado una serie de clases empaquetadas en un paquete llamado `programacionavanzada`, el nombre recomendado sería: `es.umh.programacionavanzada`, es decir, como encabezado de nuestras clases tendríamos que escribir:

```
1 package es.umh.programacionavanzada;
2
3 class MiClase {
4     ...
5 }
```

Podría haber conflicto si dos programadores de la UMH utilizaran un mismo nombre de paquete; esto se solucionaría llegando a algún acuerdo, por ejemplo, utilizar el nombre del departamento o la dirección de correo: `package es.umh.pgarrido.programacionavanzada;`.

Para terminar, podría ser que al utilizar nuestra URL en orden inverso obtengamos un nombre de paquete no válido, por ejemplo si comienza por un número, o utiliza alguna palabra reservada de Java. En tales casos, simplemente podríamos escribir un subrayado al principio.

Almacenamiento de paquetes

El hecho de que una clase pertenezca a un paquete obliga a que su archivo se almacene en un directorio con su mismo nombre, tanto el correspondiente al código fuente (`.java`) como el código compilado (`.class`). En caso de que el nombre del paquete sea un nombre compuesto (varios subpaquetes anidados), deberá existir una jerarquía de subdirectorios que se corresponda exactamente con los mismos nombres que cada uno de los subpaquetes.

Supongamos que tenemos una serie de clases `MiClase1`, `MiClase2`, ... que pertenecen a un mismo paquete, escritas en archivos con el mismo nombre que su clase pero con la extensión `.java`, que una vez compilados tendrán la extensión `.class`:

Copyright © 2012 P. Pablo Garrido Abenza

```java
// Archivo MiClase1.java
package es.umh.programacionavanzada;

class MiClase1 {
    ...
}
```

```java
// Archivo: MiClase2.java
package es.umh.programacionavanzada;

class MiClase2 {
    ...
}
```

En este caso, las clases pertenecen al paquete `es.umh.programacionavanzada`, y esto obliga a que los archivos `.java` y `.class` tengan que estar dentro de un directorio llamado `programacionavanzada`, y éste, a su vez, dentro de otro directorio llamado `umh`, y éste, finalmente, dentro de otro llamado `es`. Es decir, el nombre totalmente cualificado de una clase debe coincidir con la estructura de directorios que lo almacena, sustituyendo el punto por el carácter separador de directorios del sistema ('\' en Windows, o '/' en sistemas de tipo Unix). Por ejemplo, en Linux el archivo `MiClase1.java` se almacenaría en el siguiente directorio:

- Nombre totalmente cualificado: `es.umh.programacionavanzada.MiClase1`
- Ruta de almacenamiento: `.../es/umh/programacionavanzada/MiClase1.java`

Los archivos `.java` y `.class` no tienen por qué estar exactamente en la misma ruta, pero ambos deben cumplir lo anterior; de hecho, el entorno NetBeans los almacena en directorios separados. Esto permite poder distribuir clases sin tener que acompañarlas de su código fuente. Si utilizásemos NetBeans, los archivos del ejemplo anterior se almacenarían en los siguientes subdirectorios de nuestro proyecto (PRJ):

- Archivos fuente (.java): `PRJ/src/es/umh/programacionavanzada/*.java`
- Archivos compilados (.class): `PRJ/build/classes/es/umh/programacionavanzada/*.class`

Sin embargo, el directorio raíz del paquete, en nuestro caso el directorio `es`, puede estar en cualquier parte del sistema de archivos, no hay ninguna restricción, salvo que esté localizable como una ruta especificada en la variable de entorno `CLASSPATH`. Esta variable contiene una lista de rutas en las que buscar archivos `.class` o paquetes, separadas por punto y coma `';'` en Windows, o por dos puntos `':'` en caso de sistemas Unix. Por tanto, podemos almacenar o instalar paquetes de clases Java en distintas rutas, siempre y cuando se incluyan en el valor de la variable `CLASSPATH`.

Continuando con el ejemplo anterior, si finalmente queremos almacenar todos los paquetes de clases desarrollados en un único directorio compartido por todos los usuarios de un servidor, podríamos crear un directorio `/home/shared` al que tengan acceso los usuarios que las necesiten. En ese caso, podríamos copiar los archivos `.class` al directorio: `/home/shared/es/umh/programacionavanzada`.

De este modo, se tendría que incluir el directorio `/home/shared` en la variable `CLASSPATH` para que el intérprete Java (JVM) pueda encontrarlas, es decir, se incluiría el directorio que contiene al paquete, no el directorio raíz del paquete (`es`), ni el que contiene los archivos `.class`; esto último sería válido sólo para el caso de que una clase no utilice nombre de paquete, es decir, que esté en el paquete predeterminado. En el siguiente apartado se explica cómo establecer el valor de la variable `CLASSPATH`.

Copyright © 2012 P. Pablo Garrido Abenza

Variable de entorno CLASSPATH

Dependiendo de nuestro sistema, la forma de establecer el valor de la variable `CLASSPATH` cambia. En cualquier caso, esto se puede hacer de forma global por el administrador del sistema para que cualquier usuario pueda utilizarlas, o bien, de forma local por cada usuario concreto que necesite utilizar estas clases (por ejemplo, otros programadores o los usuarios de cierto programa que las use).

En el sistema **Windows**, para establecer el valor de la variable `CLASSPATH` haremos lo siguiente. Supongamos que el directorio donde hemos almacenado las clases es `C:\Compartido`:

- Desde el Panel de Control: abrir el *Panel de Control* y después *Sistema*. En la solapa *Avanzado* pulsaremos el botón *Variables de entorno*. Si la variable `CLASSPATH` no existe la añadimos, y si ya existe modificaremos su valor, que será la ruta o rutas a añadir, separadas por punto y coma `';'`. Es conveniente incluir siempre un directorio `'.'`, que hace referencia al directorio actual. Por tanto, el valor a establecer sería: `.;C:\Compartido`. En caso de utilizar la ventana del DOS para ejecutar los programas, es necesario cerrarla y volver a abrir para que los cambios tengan efecto.
- Desde la ventana del DOS: tendremos que modificar el valor mediante la orden `set`, que también nos permite consultar su valor previamente. Esto también sería válido para versiones antiguas de Windows, y mientras no se cierre la ventana; cuando se vuelva a abrir tendremos que volver a establecer.

```
C:\> set CLASSPATH                  // Consultar  el valor
C:\> set CLASSPATH=.;C:\Compartido  // Establecer el valor
```

En sistemas de tipo `Unix` (Linux, MacOS X, ...), utilizando la consola podemos establecer el valor de la variable `CLASSPATH` de forma temporal o permanente (modificando algún fichero con la misma orden). La forma de hacerlo y fichero a modificar para hacer el cambio permanente depende del *shell* o interprete de órdenes utilizado (bash, ksh, ...). Supongamos que el directorio donde reside el paquete es `/home/shared`:

- Linux (shells: bash, ksh, zsh, sh): `$HOME/.bash_profile` o `$HOME/.profile`:

```
CLASSPATH=.:$CLASSPATH:/home/shared
export PATH
```

- Linux (shells: csh, tcsh): `$HOME/.login`:

```
setenv CLASSPATH .:$CLASSPATH:/home/shared
```

Comentar que en la variable `CLASSPATH`, además de rutas también es posible especificar nombres de ficheros con la extensión `.jar` y `.zip`. Estos son archivos comprimidos que contienen un conjunto de clases (por ejemplo, un paquete) y otros recursos (iconos o sonidos), y pueden ser creados con la utilizad `jar` incluida en el JDK o con algún compresor que genere archivos `.zip`. Si se utiliza el entorno de desarrollo NetBeans, al construir el proyecto se genera el correspondiente archivo `.jar` en el subdirectorio `dist` del proyecto.

Copyright © 2012 P. Pablo Garrido Abenza

Una alternativa a modificar la variable `CLASSPATH` sería utilizando el parámetro `-classpath` (o su equivalente `-cp`) a la hora de invocar al compilador (`javac`) o al intérprete de Java (`java`) desde la línea de órdenes. El siguiente ejemplo compila y ejecuta desde la consola de Linux un programa que hace uso del paquete `es/umh/programacionavanzada`, almacenado en el directorio `/home/shared`:

```
javac -classpath "/home/shared" MiPrograma.java
java  -classpath "/home/shared" MiPrograma
```

```
javac -cp "/home/shared" MiPrograma.java
java  -cp "/home/shared" MiPrograma
```

Al utilizar el parámetro `-classpath` lo que hacemos es sobreescribir el valor que pudiéramos tener en la variable de entorno `CLASSPATH`, por lo que puede ser útil para poder utilizarlo en algún momento puntual, sin tener que modificar de forma permanente la configuración del sistema. El valor que se especifica es idéntico al que se utiliza en la variable `CLASSPATH`, es decir, una lista de rutas separadas por `';'` o `':'` (dependiendo del sistema), o archivos `.jar` o `.zip` conteniendo archivos `.class`.

Copyright ©2012- P. Pablo Garrido Abenza

5.2. Herencia

En este apartado se define el concepto de herencia, y se presenta la jerarquía de clases de Java y su clase raíz `Object`. Se explica la sobrecarga (*overload*) y sobrescritura (*override*) de métodos. Para terminar, se amplía el uso de modificadores de acceso a los miembros de una clase, como el modificador protegido (`protected`) y a nivel de paquete, así como el modificador `final`, aplicado a clases, métodos y variables.

Durante el desarrollo, muchas veces nos damos cuenta de que algunas clases tienen muchas cosas en común (datos, métodos, etc.). Sin embargo, se han escrito en clases diferentes porque hay ciertas diferencias. Por tanto, tendrán buena parte del código exactamente igual. La Programación Orientada a Objetos tiene una propiedad llamada **herencia**, que permite que unas clases hereden de otras, de tal manera que si dos clases tienen muchas cosas en común, podría escribirse una tercera clase que tuviera justamente las cosas comunes a ambas clases, y las dos clases originales heredarían de ella. La nueva clase sería la superclase, mientras que las dos clases serían subclases de ella, tambiéen denominadas clases derivadas. Con esto, todo el código que tenga la superclase ya no sería necesario reescribirlo en las dos subclases, pudiendose eliminar. Por tanto, con el uso de herencia, la ventaja más evidente es el ahorro de código duplicado en diferentes clases, es decir, se facilita la reutilización de código, lo cual, a su vez, redunda en que se facilite todo el ciclo de desarrollo: implementación, depuración y mantenimiento.

Para indicar que una clase hereda de otra se utiliza la palabra reservada `extends`. Por ejemplo, para indicar que la clase `MiClase` hereda de otra llamada `SuperClase` escribiriamos:

```
// Archivo Superclase.java
class SuperClase {
    ...
}
```

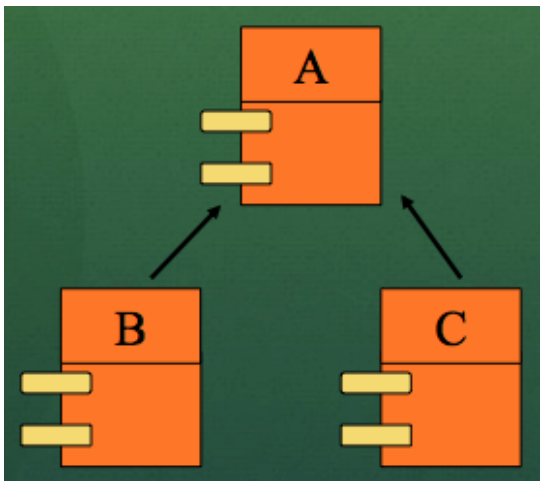
```
// Archivo MiClase.java
class MiClase extends SuperClase {
    ...
}
```

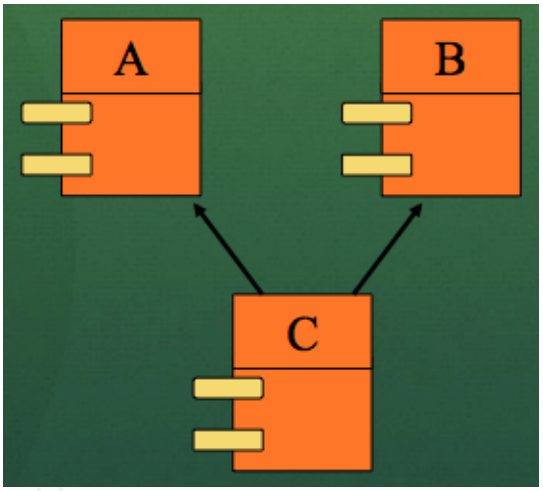
Con esto estamos consiguiendo que todas las variables, constantes y métodos de la clase `SuperClase` estén también en la subclase `MiClase`, y en cualquier otra posible subclase que también herede de `SuperClase`. Sin embargo, los constructores no se heredan, aunque pueden invocarse mediante `super()`, tal y como se explicará un poco más adelante en la sección 5.4.

Copyright © 2012 P. Pablo Garrido Abenza

Herencia simple vs. múltiple

El lenguaje Java tan sólo permite la **herencia simple**, es decir, cualquier clase tiene una y solo una clase padre (superclase). Otros lenguajes como C++ permiten la **herencia múltiple**, pero esto puede ser una fuente de problemas, por lo que se en Java se ha eliminado esta posibilidad. Por tanto, al utilizar la clásula `extends` tan sólo se puede especificar una única superclase. El siguiente ejemplo de la izquierda sería válido, pero el de la derecha no:






```java
// Herencia simple (valida)
class A {
    ...
}

class B extends A {
    ...
}

class C extends A {
    ...
}
```

```java
// Herencia multiple (no valida)
class A {
    ...
}

class B {
    ...
}

class C extends A, B {
    ...
}
```

Copyright © 2012 P. Pablo Garrido Abenza

Operador `instanceof`

El operador `instanceof` permite determinar en tiempo de ejecución si un objeto es instancia de cierta clase. Por ejemplo, para averiguar si un objeto `obj` es instancia de la clase `Circulo` escribiríamos:

```
Circulo obj = new Circulo();
// ...
if (obj instanceof Circulo) { ... }
```

Viendo el ejemplo anterior, aparentemente no tiene mucha utilidad, puesto que ya habíamos declarado que el objeto `obj` era de la clase `Circulo`. ¿Para qué nos puede interesar comprobarlo en tiempo de ejecución?.

Su principal uso es cuando se utiliza herencia, esto es, cuando se tiene una jerarquía de clases, y queremos averiguar de que subclase concreta es instancia un objeto. Antes de hacer un casting explícito entre clases, debemos asegurarnos de que el objeto a convertir es instancia-de la clase a la cual vamos a convertir; en otro caso podría obtenerse un error. Imaginemos una clase `Triangulo` y otra `Circulo`, las cuales heredan de una clase `Fig2D`. Sin entrar en detalles sobre su implementación, esta jerarquía se codificaría del siguiente modo:

```
class Fig2D { }
class Triangulo extends Fig2D { }
class Circulo   extends Fig2D { }
```

El siguiente programa utiliza la jerarquía de clases anterior, creando algún objeto de sus clases y comprobando a qué clases pertenecen con el operador `instanceof`:

```
class InstanceofDemo {
    public static void main (String[] args) {
        Fig2D     f = new Fig2D();
        Triangulo t = new Triangulo();
        Circulo   c = new Circulo();
        Circulo   n = null;  // No creado

        System.out.println ("f instanceof Object?   : " + (f instanceof Object)
            );
        System.out.println ("f instanceof Fig2D?    : " + (f instanceof Fig2D))
            ;
        System.out.println ("f instanceof Triangulo?: " + (f instanceof
            Triangulo));
        System.out.println ("f instanceof Circulo?  : " + (f instanceof Circulo
            ));

        System.out.println ("t instanceof Object?   : " + (t instanceof Object)
            );
        System.out.println ("t instanceof Fig2D?    : " + (t instanceof Fig2D))
            ;
```

Copyright © 2012 P. Pablo Garrido Abenza

```
        System.out.println ("t instanceof Triangulo?: " + (t instanceof
            Triangulo));
        //System.out.println ("t instanceof Circulo?  : " + (t instanceof
            Circulo));

        System.out.println ("c instanceof Object?    : " + (c instanceof Object)
            );
        System.out.println ("c instanceof Fig2D?     : " + (c instanceof Fig2D))
            ;
        //System.out.println ("c instanceof Triangulo?: " + (c instanceof
            Triangulo));
        System.out.println ("c instanceof Circulo?   : " + (c instanceof Circulo
            ));

        System.out.println ("n instanceof Circulo?   : " + (n instanceof Circulo
            ));
    }
}
```

```
f instanceof Object?   : true
f instanceof Fig2D?    : true
f instanceof Triangulo?: false
f instanceof Circulo?  : false
t instanceof Object?   : true
t instanceof Fig2D?    : true
t instanceof Triangulo?: true
c instanceof Object?   : true
c instanceof Fig2D?    : true
c instanceof Circulo?  : true
n instanceof Circulo?  : false
```

Las **conclusiones** que se han observado a la vista del resultado de la ejecución del programa anterior son:

- Que el operador `instanceof` sólo es correcto utilizarlo en clases que estén relacionadas por herencia, ya sean superclases o subclases, de lo contrario, obtenemos el error de compilación *Inconvertible types*; el programa tiene comentadas dos líneas debido a eso.
- Cualquier objeto se considera instancia también de sus superclases, incluyendo la clase raíz `Object`, pero a la inversa no es cierto, es decir, un objeto de una clase no es instancia de ninguna de sus subclases.
- Un objeto nulo (`null`) no es instancia de ninguna clase.

Un ejemplo más útil que el anterior sería el siguiente, en el que se define un *array* de la superclase `Fig2D`, admitiendo, por tanto, objetos de la clase `Fig2D`, así como de cualquier subclase de ella (`Triangulo` o `Circulo`). Luego recorre el *array* mostrando y contando cuántos objetos hay de cada clase. De nuevo puede comprobarse que los objetos que no están creados (`null`) no son instancia de ninguna clase, aunque se hayan declarado de alguna clase concreta.

Copyright © 2012 P. Pablo Garrido Abenza

```
class Fig2D { }
class Triangulo extends Fig2D { }
class Circulo   extends Fig2D { }

public class InstanceofDemo2 {
    public static void main (String[] args) {
        int contTriangulos, contCirculos, contNulos;
        Fig2D[]   lista;
        Triangulo t1, t2, t3, t4;
        Circulo   c1, c2, c3, c4;

        // Creamos 3 triangulos y 3 circulos
        t1 = new Triangulo();
        t2 = new Triangulo();
        t3 = new Triangulo();
        t4 = null;  // No creado
        c1 = new Circulo();
        c2 = new Circulo();
        c3 = new Circulo();
        c4 = null;  // No creado

        // Almacenamos todas las referencias en el array
        lista = new Fig2DInstanceofDemo2[8];
        lista[0] = t1;  // Asignamos un objeto triangulo
        lista[1] = t2;  // Asignamos un objeto triangulo
        lista[2] = t3;  // Asignamos un objeto triangulo
        lista[3] = t4;  // Asignamos un objeto nulo
        lista[4] = c1;  // Asignamos un objeto circulo
        lista[5] = c2;  // Asignamos un objeto circulo
        lista[6] = c3;  // Asignamos un objeto circulo
        lista[7] = c4;  // Asignamos un objeto nulo

        // Mostramos y contamos los objetos de la lista
        contTriangulos = contCirculos = contNulos = 0;
        for (int i=0; i<lista.length; i++) {
           System.out.println ("Objeto " + i + " = " + lista[i]);
           if (lista[i] instanceof Triangulo) {
               contTriangulos++;
           } else if (lista[i] instanceof Circulo) {
               contCirculos++;
           } else {
              contNulos++;
           }
        }
        System.out.println ("Total " + contTriangulos + " triangulos, " +
```

Copyright © 2012 P. Pablo Garrido Abenza

```
                                        contCirculos    + " circulos, y " +
                                        contNulos       + " nulos");
    }
}
```

```
Objeto 0 = Triangulo@6345e044
Objeto 1 = Triangulo@86c347
Objeto 2 = Triangulo@f7e6a96
Objeto 3 = null
Objeto 4 = Circulo@3487a5cc
Objeto 5 = Circulo@35960f05
Objeto 6 = Circulo@eb42cbf
Objeto 7 = null
Total 3 triangulos, 3 circulos, y 2 nulos
```

Copyright ©2012- P. Pablo Garrido Abenza

Jerarquía de clases - Clases `Object` y `Class`

Mediante la herencia se crea una jerarquía de clases (ver figura 5.2), a la que pertenecen tanto las propias clases del lenguaje Java como todas las clases desarrolladas por nosotros. La clase raíz de esta jerarquía es la clase `Object`. Cualquier clase hereda de la clase `Object`, de forma directa o indirectamente; cuando una clase no especifica que hereda de ninguna clase, esto es, no utiliza la claúsula `extends`, entonces hereda directamente de la clase `Object`.

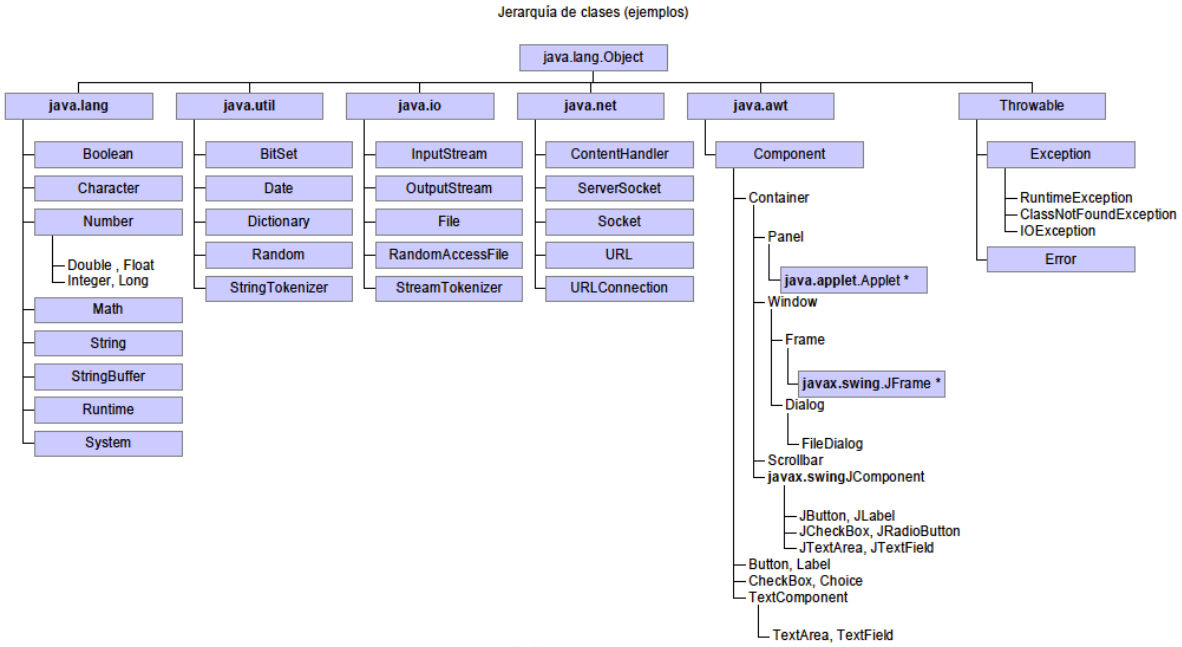


Figura 5.2: Jerarquía de clases de Java - Clase raíz `Object`

Clase `Object`

Como se acaba de decir, la clase `Object` es la clase raíz de la jerarquía de clases de Java, la cual se define en el paquete `java.lang`. Aunque esta clase tiene un **constructor** (sin parámetros), en realidad nunca se crean objetos de esta clase, sino que se utiliza como un conjunto de métodos que automáticamente están disponibles en cualquier otra clase.

Asimismo, dispone un **finalizador** (`finalize()`), que será el que se ejecute cuando el recolector de basura vaya a eliminar cualquier objeto de cualquier clase, salvo para las clases en las que el método finalizador haya sido sobreescrito. Sin embargo, este finalizador no realiza ninguna acción, tiene código vacío.

Los **métodos** más habituales son los que se muestran en la tabla 5.1. De ellos hay que destacar el método `equals()` y el método `toString()`. Aunque ambos métodos son heredados por todas las clases, es necesario sobreescribirlos para que tengan realmente utilidad. Cuando se invoca alguno de estos métodos mediante algún objeto, primero se comprueba si se ha sobreescrito en la clase de la que es instancia el objeto, en caso contrario se comprobará si su superclase lo tiene, y así sucesivamente hasta llegar a la clase raíz `Object`, realizando las acciones definidas por defecto en esta clase raíz, lo cual no suele ser lo que se quiere.

Copyright © 2012 P. Pablo Garrido Abenza

Método	Descripción
`clone()`	Crea una copia del objeto actual. Para poder utilizarlo es necesario implementar la interface `Cloneable` (esto se explica al final de este capítulo).
`equals()`	Compara si el objeto actual es igual que el objeto que recibe como argumento, retornando `true` si son iguales y `false` en caso contrario. Es necesario sobreescribir este método, ya que por defecto se comporta igual que si comparásemos los dos objetos con el operador `'=='`, es decir, compararíamos si ambos son referencias del mismo objeto.
`getClass()`	Retorna una referencia a la clase de un objeto, es decir, retorna un objeto de la clase `Class`, la cual se explica en este mismo apartado.
`hashCode()`	Retorna el identificador único del objeto actual. Cada vez que se crea una instancia de una clase se le asigna un identificador único (referencia).
`toString()`	Retorna una representación en forma de cadena del objeto actual. Al igual que el método equals(), la mayoría de las clases que escribamos deberán sobreescribir este método, puesto que su comportamiento por defecto es construir una cadena formada por el nombre de la clase, un carácter arroba `'@'` y su referencia (valor *hash*).

Cuadro 5.1: Clase `Object` - Algunos métodos

La función `toString()` de la clase `Object` retorna una representación en forma de cadena de un objeto, pero es necesario modificar su comportamiento sobreescribiendo siempre esta función. La forma de sobreescribir la función `toString()` consiste en escribir una función exactamente como se muestra a continuación, que en su interior construye una cadena de caracteres `String` concatenando mediante el operador `'+'` o `'+='` cadenas de caracteres constantes, variables de clase, el resultado de algún método, etc. Esta cadena retornada se suele utilizar para imprimir objetos mediante el método `System.out.println(...)`. Por ello, la cadena puede contener caracteres de salto de línea como: `'\r'`, `'\n'`, ...

```
class MiClase {
   int v;

   public String toString () {
      String s = "Valor: " + v;
      return (s);
   }
}
```

En caso de que una clase no sobreescriba el método `toString()`, se hereda la existente en la clase `Object`, la cual retorna una cadena formada por el nombre de la clase, un carácter arroba `'@'` y su referencia (valor *hash* único del objeto); más adelante se mostrará un ejemplo. En concreto, la siguiente función `toStringMia()` sería equivalente a la ya implementada en la clase `Object`:

```
public String toStringMia () {
   String s = this.getClass().getName() + '@' +
              Integer.toHexString(this.hashCode());
   return (s);
}
```

Copyright © 2012 P. Pablo Garrido Abenza

Por otro lado, el método `equals()` es un método definido por el usuario en cada clase que le permite decidir cuándo dos objetos de una misma clase (instancias) son iguales, según el concepto de igualdad que quiera definir, retornando un valor booleano (`true` o `false`). Normalmente se comparan las variables de instancia propias de cada objeto (estado), pero puede que todas ellas, o sólo las que nos interesen. De forma similar al método `toString()`, todas las clases deberían sobreescribir este método, ya que en caso contrario se utilizaría el método `equals()` heredado de la clase `Object`. El resultado de utilizar dicho método `equals()` sería exactamente el mismo que si utilizásemos el operador `'=='`, es decir, estaríamos comparando si son referencias del mismo objeto, no su contenido. Algo así:

```
public boolean equals (Object o) {
   return ( (this == o) );
}
```

La forma de sobreescribir este método sería escribiendolo exactamente con el mismo prototipo que se muestra a continuación, aunque lógicamente con código diferente según las variables a comparar.

```
class MiClase {
   int v;

   public boolean equals (Object o) {
      MiClase otro = (MiClase) o;
      if (this.v == otro.v) {
         return (true);
      } else {
         return (false);
      }
   }
}
```

Entonces, ¿qué dos objetos estamos comparando?. Respuesta: uno es el objeto con el que invocamos al método `equals()`, y otro es el que le pasamos como argumento que dentro del método `equals()` o que finalmente llamamos `otro`, siendo indistinto en qué orden se especifiquen. Por ello, para acceder a las distintas variables de instancia de ambos objetos para compararlas, para el primer objeto utilizamos `this` (el objeto actual), y para el segundo utilizamos la referencia `otro`. Por ejemplo, la forma de utilizar este método para comparar dos objetos `m1` y `m2` sería cualquiera de las siguientes, ambas equivalentes:

```
1 MiClase m1 = new MiClase(...);
2 MiClase m2 = new MiClase(...);
3 ...
4 if ( m1.equals(m2) ) {
5    // Son iguales
6 }
```

```
1 MiClase m1 = new MiClase(...);
2 MiClase m2 = new MiClase(...);
3 ...
4 if ( m2.equals(m1) ) {
5    // Son iguales
6 }
```

Copyright © 2012 P. Pablo Garrido Abenza

La forma anterior de escribir la función `equals()` es una forma simplificada, que no es robusta para ciertas situaciones que se pudieran producir. Por ejemplo, ¿qué pasaría si el objeto que se pasa como argumento es `null`?, ¿y si es de una clase distinta?, ¿podría optimizarse para el caso de que ambos objetos fuesen una referencia al mismo objeto? (sin llegar a comparar el contenido). Por tanto, la función `equals()` debería contemplar siempre esos casos, y escribirse siempre de la siguiente forma, modificando sólo la última parte que compara el contenido:

```
class MiClase {
   int v;

   public boolean equals (Object o) {
      // ¿Objeto nulo?
      if (o == null) return false;
      // ¿Misma clase? (si o fuese null retorna false)
      if (!(o instanceof MiClase)) return false;
      // ¿Misma referencia?  (idem que ==)
      if (this == o) return true;
      // ¿Mismo contenido?
      MiClase otro = (MiClase) o;
      if (this.v == otro.v) {
         return (true);
      } else {
         return (false);
      }
   }
}
```

Respecto al parámetro de la clase `Object` que recibe el método `equals()`:

- El haber escrito que este parámetro es de la clase `Object` nos obliga a convertirlo (mediante *casting*) de la clase genérica `Object` a la clase `MiClase`, en un objeto que hemos llamado `otro`. Si no hiciéramos esta conversión, e intentásemos acceder a la variable `v` directamente desde el objeto `o`, puesto que éste es de la clase `Object`, se intentaría buscar una variable llamada `v` en la clase `Object`, la cual, obviamente no existe, generando un error de ejecución.

- Podría caerse en la tentación de especificar que el argumento `o` sea de la clase `MiClase`, y nos evitaríamos tener que realizar la conversión, ni comprobar si son de la misma clase con `instanceof`. Sin embargo, al hacer esto no estaríamos sobreescribiendo el método `equals()` heredado de la clase `Object`, sino que sería otro método distinto, con el mismo nombre pero con parámetros diferentes (polimorfismo). Esto funcionaría para los ejemplos expuestos en este apartado, pero hay ocasiones en las que se tienen colecciones de objetos (lo cual se verá en la siguiente unidad), y alguna clase llama automáticamente al método `equals()` que recibe un `Object`, ejecutándose por tanto el método que se hereda de la clase `Object`, e ignorándose el método `equals()` que hemos escrito en nuestra clase `MiClase`.

Copyright © 2012 P. Pablo Garrido Abenza

En el siguiente **ejemplo** tenemos las clases `Circulo` y `Triangulo`, en las que vamos a utilizar los métodos `equals()` y `toString()`. Para comprobar el efecto de no sobreescribir estas funciones, en la clase `Triangulo` sí que lo hemos hecho, pero en la clase `Circulo` lo hemos olvidado. El programa de prueba crea dos objetos de cada clase con los mismos valores (estado). Finalmente los imprime por pantalla y compara los objetos; veremos que al compararlo con `equals()` el resultado será `true` si se ha sobreescrito correctamente el método `equals()`, mientras que al compararlos con el operador `'=='` el resultado será siempre `false`, pues son objetos distintos (creados con `new`). Como se había comentado previamente, al intentar imprimir cualquier objeto se invoca automáticamente al método `toString()` de su clase para convertirlo a cadena.

```
1  class Circulo {
2     // Datos: radio del circulo
3     public double radio;
4
5     // Constructor
6     public Circulo (double radio) {
7        this.radio = radio;
8     }
9
10    // Upss! Hemos olvidado sobreescribir los metodos equals() y toString();
11    // se utilizaran los que se heredan de la clase Object
12 }
13
14 class Triangulo {
15    // Datos: longitud de los 3 lados del triangulo
16    public double a,b,c;
17
18    // Constructor
19    public Triangulo (double a, double b, double c) {
20       this.a = a;  this.b = b;  this.c = c;
21    }
22
23    // Sobreescribimos el metodo equals() heredado de Object
24    // Consideramos que dos triangulos son iguales si todos sus lados lo son.
25    public boolean equals (Object o) {
26       Triangulo otro = (Triangulo) o;
27       if ((this.a == otro.a) && (this.b == otro.b) && (this.c == otro.c)) {
28          return (true);
29       } else {
30          return (false);
31       }
32    }
33
34    // Sobreescribimos el metodo toString() heredado de Object
35    public String toString () {
36       String s = "Esto es un triangulo de lados " +
```

Copyright © 2012 P. Pablo Garrido Abenza

```
37                  "a=" + a + " b=" + b + " c=" + c;
38        return (s);
39     }
40  }
41
42  public class ObjectDemo {
43     public static void main (String args[]) {
44        Circulo   c1, c2;
45        Triangulo t1, t2;
46
47        // Creamos dos circulos con identicos valores
48        c1 = new Circulo (20.0);
49        c2 = new Circulo (20.0);
50        // Mostramos y comparamos los circulos
51        System.out.println ("El objeto c1 es: " + c1);  // c1.toString()
52        System.out.println ("El objeto c2 es: " + c2);  // c2.toString()
53        System.out.println ("Comparacion con equals(): " + c1.equals(c2));
54        System.out.println ("Comparacion con oper. ==: " + (c1 == c2));
55
56        // Creamos dos triangulos con identicos valores
57        t1 = new Triangulo (10.0,10.0,10.0);
58        t2 = new Triangulo (10.0,10.0,10.0);
59        // Mostramos y comparamos los triangulos
60        System.out.println ("El objeto t1 es: " + t1);  // t1.toString()
61        System.out.println ("El objeto t2 es: " + t2);  // t2.toString()
62        System.out.println ("Comparacion con equals(): " + t1.equals(t2));
63        System.out.println ("Comparacion con oper. ==: " + (t1 == t2));
64     }
65  }
```

Si ejecutamos el programa anterior nos daremos cuenta de varias cosas: (1) que al imprimir los objetos de la clase `Circulo` obtenemos algo extraño, resultado de haber olvidado sobreescribir el método `toString()` en dicha clase, mientras que al imprimir objetos de la clase `Triangulo`, se muestran según la cadena `String` que construimos, cada objeto con sus valores, (2) que al comparar los circulos siempre obtenemos `false`, tanto al utilizar el método `equals()` como el operador `'=='`, resultado de haber olvidado sobreescribir el método `equals()` en dicha clase.

```
El objeto c1 es: Circulo@50a9ae05
El objeto c2 es: Circulo@33dff3a2
Comparacion con equals(): false
Comparacion con oper. ==: false
El objeto t1 es: Esto es un triangulo de lados a=10.0 b=10.0 c=10.0
El objeto t2 es: Esto es un triangulo de lados a=10.0 b=10.0 c=10.0
Comparacion con equals(): true
Comparacion con oper. ==: false
```

Copyright © 2012 P. Pablo Garrido Abenza

Una alternativa al uso de `instanceof` en el método `equals()` para comprobar si la instancia recibida como argumento es de la clase actual podría ser comparando el objeto de la clase `Class` que retorna el método `getClass()` de los dos objetos a comparar; la clase `Class` se describe en la siguiente sección. Sin embargo, se recomienda seguir utilizando el operador `instanceof`, entre otras cosas, porque para utilizar `getClass()` es necesario comprobar previamente si el objeto es `null`, cosa que con `instanceof` no era necesario (aunque lo habíamos hecho igualmente).

```
class MiClase {
   int v;

   public boolean equals (Object o) {
      // ¿Objeto nulo?
      if (o == null) return false;
      // ¿Misma clase?
      if (this.getClass() != o.getClass()) return false;
      ...
   }
}
```

Copyright © 2012 P. Pablo Garrido

Conclusiones:

1. Cuando una clase no sobreescribe el método `equals()`, cuando se invoque se utilizará el método heredado de la clase `Object`, que se comporta exactamente que si utilizásemos el operador `'=='`, es decir, estamos comparando si son referencias del mismo objeto, no su contenido.

2. El método `toString()` se invoca automáticamente si se intenta imprimir un objeto por pantalla.

3. Cuando una clase no sobreescribe el método `toString()`, cuando se invoque, por ejemplo al intentar imprimir una instancia de dicha clase, se comprobará si su superclase lo tiene, y así sucesivamente hasta llegar a la clase raíz `Object`, que como sabemos, sí que la define, aunque de una forma que lo más probable es que no sea lo que queramos.

Clase `Class`

La clase `Class` tiene una serie de métodos que permite obtener cierta información sobre una clase como su nombre (`getName()`), si es abstracta, si se corresponde con un tipo primitivo, etc. (ver tabla 5.2).

Método	Descripción
`getName()`	Retorna el nombre de la clase del objeto `Class`.
`getPackage()`	Retorna el nombre del paquete al que pertenece la clase del objeto `Class`.
`getSuperclass()`	Retorna un objeto `Class` que representa a la superclase de la clase que representa un objeto `Class`.

Cuadro 5.2: Clase `Class` - Algunos métodos

El siguiente programa utiliza la clase `Class` para obtener el nombre del programa que se está ejecutando, y muestra de nuevo el resultado de utilizar la función `toString()` sin reescribirla en nuestras clases. Si se recuerda de temas anteriores, el lenguaje C tenía una forma de obtener esta información, ya que el primer elemento del array de argumentos que recibe la función `main(int argc, char *argv[])` era siempre el nombre del programa (`argv[0]`). En el caso del lenguaje Java, no era así, ya que el primer elemento del array equivalente no era el nombre del programa sino el primer argumento recibido. Luego, utiliza también la clase `Class` para obtener el nombre de la clase (`getName()`) de tres objetos de tres clases distintas (`String`, `Circulo`, `Triangulo`), y en cada caso de una forma diferente; vemos que dado un objeto o de cualquier clase, podrían encadenarse llamadas a métodos de la forma `o.getClass().getName()`, evitando tener que declarar algún objeto intermedio.

```
1  class Circulo {
2     // ...
3  }
4
5  class Triangulo {
6     // ...
7  }
8
9  public class ClassDemo {
10    public static void main (String args[]) {
11       Class c = ClassDemo.class;
12       System.out.println ("Este programa se llama: " + c.getName());
13
14       String o1 = new String("Hola");
15       c = o1.getClass();
16       System.out.println ("Objeto o1 es de la clase: " + c.getName());
17       System.out.println ("Objeto o1 como cadena es: " + o1.toString());
18
19       Circulo o2 = new Circulo();
20       System.out.println ("Objeto o2 es de la clase: " + o2.getClass().getName
             ());
21       System.out.println ("Objeto o2 como cadena es: " + o2.toString());
22
23       Triangulo o3 = new Triangulo();
24       imprimeObjeto (o3);
25    }
26
27    private static void imprimeObjeto (Object o) {
28        System.out.println ("Objeto o? es de la clase: " + o.getClass().getName
              ());
29        System.out.println ("Objeto o? como cadena es: " + o.toString());
30    }
31 }
```

Copyright © 2012 P. Pablo Garrido Abenza

Este programa genera el siguiente resultado al ejecutarlo desde la consola:

```
Este programa se llama: ClassDemo
Objeto o1 es de la clase: java.lang.String
Objeto o1 como cadena es: Hola
Objeto o2 es de la clase: Circulo
Objeto o2 como cadena es: Circulo@33dff3a2
Objeto o? es de la clase: Triangulo
Objeto o? como cadena es: Triangulo@33dff3a2
```

Otro aspecto destacable de este ejemplo es el uso de la función `imprimeObjeto()`, que recibe como argumento un objeto de la clase `Object` (línea 29). Como se puede ver, lo hemos invocado con el objeto `o3` de la clase `Triangulo` (línea 26), no de la clase `Object`; esto es válido, ya que la clase `Triangulo` hereda de la clase `Object` (en este caso de forma directa), y la relación de herencia se puede leer como *es-un*, es decir, un `Triangulo` *es-un* `Object`. Lo mismo hubiera ocurrido si lo hubiéramos invocado con un objeto `Circulo`, o un objeto `String`, o cualquier otro, y sin necesidad de realizar ninguna conversión (*casting*). Es decir, que esa función serviría para no tener que repetir unas cuantas líneas de código. En otras palabras, en cualquier lugar donde se especifique que se recibe un `Object` sería válido pasar un objeto de cualquier clase, ya que cualquiera de ellas será subclase de `Object`, directa o indirectamente. A la inversa no ocurre igual, es decir, si el método `imprimeObjeto()` se hubiera declarado como que recibe un argumento de la clase `Triangulo`, sólo sería válido pasar valores de dicha clase, no de clases superiores (`Object`) o hermanas (`Circulo`).

Copyright ©2012- P. Pablo Garrido Abenza

Modificadores

Para una clase podemos utilizar los modificadores que se muestran en la tabla 5.3. Básicamente nos fijaremos en: `public`, `abstract`, y `final`; los modificadores `private` y `protected` podemos ignorarlos, ya que no son nada habituales (se utilizan con clases anidadas o internas.

Método	Descripción
`public`	La clase podrá utilizarse desde fuera del fichero donde está implementada. En un mismo archivo podemos escribir varias clases, y sólo una de ellas es pública (`public`), la que le da nombre al archivo; el resto no llevarán ningún modificador, y sólo serán visibles dentro de ese archivo (no es válido el uso de `private` en ese contexto).
`private`	Sólo es válido para clases internas, esto es, clases que se declaran dentro de otras, quedando ocultas fuera de la clase que la contiene (IGNORAR).
`protected`	Sólo es válido para clases internas, esto es, clases que se declaran dentro de otras, quedando ocultas fuera de la clase que la contiene, aunque visible desde otras subclases (IGNORAR).
`abstract`	Declara una clase abstracta, que al menos contiene un método abstracto o virtual, el cual tendrá que ser definido por alguna subclase (ver sección 5.3).
`final`	Una clase `final` significa que ninguna clase puede heredar de ella. De forma automática, todos sus métodos se convierten también en métodos `final`, es decir, no pueden sobreescribirse.

Cuadro 5.3: Modificadores de acceso para una clase

Copyright © 2012 P. Pablo Garrido

En un capítulo anterior ya se mostraron algunos modificadores de acceso de los métodos, pero hay algunos más, los cuales se resumen en la tabla 5.4.

Cuando una clase hereda de otra, hereda cualquier miembro que sea `public` o `protected`; aquellos que sean `private` sólo serán visibles en la superclase, no serán visibles en subclases.

Método	Descripción
`public`	Método visible desde cualquier otra clases, ya sea del mismo paquete o de otro paquete distinto.
`private`	Método visible únicamente dentro de la misma clase, no en subclases.
`protected`	Método visible dentro de la misma clase, y además, desde todas las subclases que hereden de ella, que estén en el mismo paquete.
`static`	Método estático que no accede a variables de instancia, tan sólo a las variables de clase (`static`) compartidas por todos los objetos de la clase. Se invocan utilizando el nombre de la clase, no un objeto (aunque también sería válido).
`abstract`	Método abstracto o virtual que tendrá que ser definido por alguna subclase. Si una clase contiene uno o más métodos abstractos, deberá declararse abstracta.
`final`	Un método `final` no podrá sobreescribirse por ninguna subclase. Si una clase es `final`, todos sus métodos lo serán. Si tenemos la seguridad de que esto será así, conviene poner el modificador `final`, pues el código se ejecutará más rápido (eliminamos un gasto extra al invocar al método, pues no se hace ninguna llamada, sino que el compilador la sustituye por el código completo de la función: técnica de inclusión de código en línea. En C++, se usaban los `inline`). Todos los métodos de una clase `final` son implícitamente `final`. Estos métodos utilizan, por tanto, ligado estático, no dinámico, que siempre es más rápido.
`synchronized`	Método sincronizado que está protegido mediante exclusión mutua para poder ejecutarse en una aplicación multihilo. Si un objeto está ejecutando este método, su ejecución no podrá ser interrumpida si desde otro hilo otro objeto intenta ejecutarlo también, teniendo que esperar a que el primer objeto termine la ejecución.
`native`	Método nativo, que permite la inclusión de código escrito en otros lenguajes, como por ejemplo C.

Cuadro 5.4: Modificadores de acceso para un método

Copyright © 2012 P. Pablo Garrido Abenza

5.3. Clases abstractas

Una clase abstracta es una clase que incluye los componentes habituales de cualquier clase (variables de clase, variables de instancia, constantes, constructores y métodos), pero que declara al menos un método abstracto o virtual, es decir, un método que se ha declarado pero que no se ha implementado (no tiene código). Por ello, no pueden crearse instancias de estas clases, tan sólo pueden ser ampliadas por otras subclases, las cuales deben definir todos los métodos abstractos; si no los definen todos, deberán declararse como abstractas de nuevo.

Mediante una clase abstracta establecemos una serie de métodos y atributos que tendrán en común una serie de subclases, aunque su comportamiento no será el mismo; dicho comportamiento (definición) deberá ser definido por cada una de las subclases. De estas subclases sí que podremos instanciar objetos.

En la figura 5.3 se muestra un ejemplo de una jerarquía de clases en la que interviene alguna clase abstracta. Las clases de niveles más altos son clases abstractas (líneas discontinuas), es decir, las clases `Figura`, `Fig2D`, y `Fig3D` tienen algún método abstracto que deberá ser implementado por todas sus subclases: `Cuadrado`, `Circulo`, `Cubo` y `Esfera`. Como se ve, la clase `Fig2D` es una clase genérica para figuras geométricas en 2D (cuadrados, círculos, ...), mienstras que la clase `Fig3D` es una clase genérica para figuras geométricas en 3D (cubos, esferas, ...).

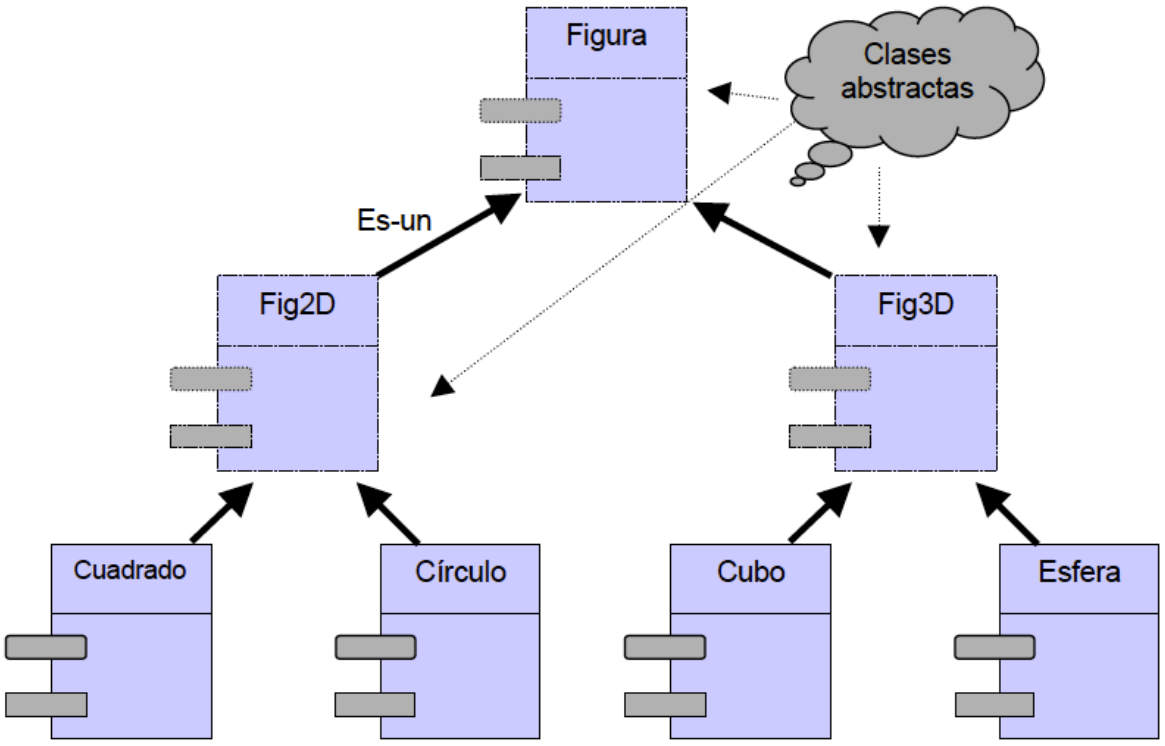


Figura 5.3: Ejemplo de jerarquía de clases

Copyright © 2012 P. Pablo Garrido Abenza

La clase `Figura` es una clase que tiene cualquier código que pueda ser común a cualquiera de las subclases (constantes, métodos, etc.), evitando así tener que reescribirlo en todas ellas. Tiene un método abstracto llamado `area()` y otro `volumen()`, pues aun no se sabe la figura concreta sobre la que se aplicarán para calcular el área o volumen, respectivamente. La clase `Fig2D` implementa el método `volumen()`, puesto que todos los objetos 2D tienen un volumen nulo, pero como aun sigue sin implementar el método `area()` debe seguir siendo abstracta. Las subclase `Fig3D` debe ser abstracta también, puesto que seguimos sin implementar los dos métodos. Serán las 4 subclases `Cuadrado`, `Circulo`, `Cubo` y `Esfera` las que finalmente terminen de implementar dichos métodos, no siendo abstractas. Esto quiere decir, que sólo podrán crearse objetos de estas 4 clases, las que no son abstractas; las otras 3 simplemente nos sirven para facilitar el desarrollo.

El **código** para las clases de la jerarquía anterior sería el siguiente:

```
import java.text.DecimalFormat;

// Clase Figura
abstract class Figura {
   // Numero total de figuras 2D + 3D
   static int numFiguras = 0;
   // Formato de los resultados a mostrar
   static DecimalFormat f = new DecimalFormat ("#,##0.00");

   abstract public double area ();
   abstract public double volumen ();

   public String toString () {
      String s = "area=" + f.format(area()) +
                 " volumen=" + f.format(volumen());
      return (s);
   }
}
```

```
// Clase Fig2D
abstract class Fig2D extends Figura {
   // Numero total de figuras 2D
   static int numFig2D = 0;

   public double volumen () {
      return (0.0) ;
   }
}
```

```
// Clase Fig3D
abstract class Fig3D extends Figura {
   // Numero total de figuras 3D
   static int numFig3D = 0;
}
```

Copyright © 2012 P. Pablo Garrido Abenza

```java
// Clase Cuadrado
class Cuadrado extends Fig2D {
   double lado;

    public Cuadrado (double lado) {
       this.lado = lado;
       numFiguras++;
       numFig2D++;
    }

    public double area () {
       return (lado*lado);
    }
}
```

```java
// Clase Circulo
class Circulo extends Fig2D {
   double radio;

   public Circulo (double radio) {
      this.radio = radio;
      numFiguras++;
      numFig2D++;
   }

   public double area () {
      return (Math.PI*radio*radio);
   }
}
```

```java
// Clase Cubo
class Cubo extends Fig3D {
   double lado ;

   public Cubo (double lado) {
      this.lado = lado;
      numFiguras++;
      numFig3D++;
   }

   public double area () {
      return (6*lado*lado);
   }

   public double volumen () {
      return (Math.pow (lado,3));
   }
}
```

```java
// Clase Esfera
class Esfera extends Fig3D {
   double radio ;

   public Esfera (double radio) {
      this.radio = radio;
      numFiguras++;
      numFig3D++;
   }

   public double area () {
      return (4*Math.PI*radio*radio
          );
   }

   public double volumen () {
      return (4*Math.PI*Math.pow(
          radio,3)/3);
   }
}
```

La clase `Figura` tiene una variable de clase (compartida) que contará el número total de figuras creadas, sean de la clase que sean, y dos métodos abstractos `area()` y `volumen()`, los cuales tendrán que ser definidos por las subclases. Del mismo modo, las clase `Fig2D` y `Fig3D` tienen otra variable de clase que contará el número de figuras 2D y 3D creadas, respectivamente, cuya suma deberá coincidir con la variable de la clase `Figura`. Todas estas variables se heredan en las subclases, incrementándose en sus respectivos constructores. Además, en la clase Fig2D ya podemos implementar el método volumen(), puesto que cualquier figura 2D tendrá un volumen nulo (0).

Copyright © 2012 P. Pablo Garrido Abenza

En cuanto a la clase `Figura`, ya se ha comentado que contiene código común a todas las subclases, para evitar tener que reescribirlo en todas ellas. Sin embargo, aunque la clase `Figuras` estuviese totalmente vacía, y no se puedan crear objetos de ella por ser abstracta, aun así tendría una utilidad: el hecho de declararla y que las otras hereden de ella nos ofrece la posibilidad de poder declarar un array cuyo tipo base sea `Figura`, el cual podrá almacenar objetos de cualquier subclase (no abstracta): `Cuadrado`, `Circulo`, `Cubo`, y `Esfera`.

```
public class AbstractDemo {
   final static int NUM_OBJETOS = 10;

   public static void main (String[] args) {
      Figura[] lista;
      Figura   f;
      double areaTotal, volumenTotal;

      // Creamos el array y lo llenamos de objetos
      lista = new Figura[NUM_OBJETOS];
      f = new Cuadrado (10.0); lista[0] = f;
      f = new Circulo (7.0);   lista[1] = f;
      f = new Cubo (12.5);     lista[2] = f;
      f = new Esfera (5.0);    lista[3] = f;
      f = new Cuadrado (15.0); lista[4] = f;

      // Mostramos los objetos y los contadores
      areaTotal = volumenTotal = 0;
      for (int i=0; i<NUM_OBJETOS; i++) {
         f = lista[i];
         System.out.print ("Elemento " + i + " es ");
         if (f == null) {
            System.out.print ("nulo");
         } else {
            System.out.print (((f instanceof Fig2D) ? "2D" : "3D"));
            areaTotal    += f.area();
            volumenTotal += f.volumen();
         }
         System.out.print (", concretamente es un ");
         if (f instanceof Cuadrado) {
            System.out.println ("cuadrado, " + f.toString());
         } else if (f instanceof Circulo) {
            System.out.println ("circulo,  " + f.toString());
         } else if (f instanceof Cubo) {
            System.out.println ("cubo,     " + f.toString());
         } else if (f instanceof Esfera) {
            System.out.println ("esfera,   " + f.toString());
         } else {
            System.out.println ("desconocido");
```

Copyright © 2012 P. Pablo Garrido Abenza

```
            }
        }
        System.out.println ( "Total Fig2D  : " + Fig2D.numFig2D);
        System.out.println ( "Total Fig3D  : " + Fig3D.numFig3D);
        System.out.println ( "Total figuras: " + Figura.numFiguras);
        System.out.println (
            "Suma area= " + Figura.f.format(areaTotal) +
            " volumen= "  + Figura.f.format(volumenTotal));
    }
}
```

```
Elemento 0 es 2D, concretamente es un cuadrado, area=100,00 volumen=0,00
Elemento 1 es 2D, concretamente es un circulo,  area=153,94 volumen=0,00
Elemento 2 es 3D, concretamente es un cubo,     area=937,50 volumen=1.953,12
Elemento 3 es 3D, concretamente es un esfera,   area=314,16 volumen=523,60
Elemento 4 es 2D, concretamente es un cuadrado, area=225,00 volumen=0,00
Elemento 5 es nulo, concretamente es un desconocido
Elemento 6 es nulo, concretamente es un desconocido
Elemento 7 es nulo, concretamente es un desconocido
Elemento 8 es nulo, concretamente es un desconocido
Elemento 9 es nulo, concretamente es un desconocido
Total Fig2D  : 3
Total Fig3D  : 2
Total figuras: 5
Suma area= 1.730,60 volumen= 2.476,72
```

Copyright ©2012- P. P

5.4. Constructores y herencia - `super`

Como ya se había comentado, **los constructores no se heredan**, aunque pueden invocarse desde las subclases. También se explicó el uso de la palabra clave `this`, que servía para dos cosas: (1) evitar conflictos de nombres entre algún argumento y alguna variable de instancia, y (2) para invocar un constructor desde otro, dentro de la misma clase.

De forma muy similar, pero relacionado con la herencia, existe la palabra reservada `super`, que también tiene dos usos:

1. **Acceder a algún miembro de la suplerclase (variable o método)**. Esto practicamente no se utiliza, ya que por la propia relación de herencia, todo lo que defina la clase padre (superclase) se hereda en la subclase, estando directamente utilizable, no es necesario hacer nada especial como el uso de `super`. Sin embargo, la subclase podría haber sobreescrito algún método de la superclase, con código diferente, y puede que en algún punto nos interesase utilizar ese método tal cual lo habríamos heredado, no como lo hemos sobreescrito; en ese caso, para evitar el conflicto de nombres podríamos utilizar `super` para hacer referencia al método de la superclase.

```
public class MiSuperClase {
   public boolean miMetodo () {
      // ...
      return (true);
   }
}

public class MiSubClase extends MiSuperClase {
   // Sobreescribimos miMetodo() que heredamos
   public boolean miMetodo () {
      // ...
      // Invocamos al miMetodo() que heredamos
      return (!super.MiMetodo());
   }
}
```

   En el caso de que la subclase no sea una subclase directa del método a invocar, es decir, que haya una jerarquía de varios niveles por medio, no podemos utilizar llamadas `super()` en cascada. Por ejemplo, para invocar a un método `f()` de una clase `Abuela` (dos niveles por encima de la clase desde donde haremos la llamada), sería incorrecto escribir: `super.super.f();` (obtendríamos un error de sintaxis). Para estas situaciones debemos utilizar `casting`: `((Abuela) this).f();`

2. **Invocar a un constructor de la superclase**. De forma similar al uso que tenía `this` para invocar un constructor desde otro de la misma clase, la palabra `super` puede utilizarse para invocar a cualquier constructor de la superclase desde un constructor de la subclase. Dependiendo del número y tipo de argumentos especificados, así se ejecutará un constructor u otro. Al igual que con `this()`, si un constructor utiliza `super()` para invocar a otro, ésta debe ser la primera línea, antes incluso que la declaración de alguna posible variable local.

Copyright © 2012 P. Pablo Garrido Abenza

```
public class MiSuperClase {
   public MiSuperClase () {
      // ... algunas cosas ...
   }
}

public class MiSubClase extends MiSuperClase {
   public MiSubClase () {
      // Invocamos al constructor de la superclase
      super ();
      // ... mas cosas ...
   }
}
```

El ejemplo de la sección anterior podría modificarse para no repetir algo de código existente en los constructores de las 4 clases no abstractas. Si nos fijamos en el constructor de las clases `Cuadrado` y `Circulo`, ambos incrementan los dos contadores de objetos, el global y el de figuras 2D; podríamos haber definido un constructor en la clase `Fig2D` con esas dos líneas de código, e invocarlo desde los constructores de las 2 subclases. De forma similar se podría hacer con la clase `Fig3D`, definiendo un constructor en ella que sea invocado mediante `super()` desde sus dos subclases `Cubo` y `Esfera`, que hacían las dos mismas asignaciones, quedando como sigue.

```
1  abstract class Fig2D extends Figura
       {
2     static int numFig2D = 0;
3
4     public Fid2D () {
5        numFiguras++;
6        numFig2D++;
7     }
8
9     public double volumen () {
10       return (0.0) ;
11    }
12 }
```

```
1  abstract class Fig3D extends Figura
       {
2     static int numFig3D = 0;
3
4     public Fid3D () {
5        numFiguras++;
6        numFig3D++;
7     }
8  }
```

Copyright © 2012 P. Pablo Garrido Abenza

```
1  class Cuadrado extends Fig2D {
2     double lado;
3
4      public Cuadrado (double lado) {
5         super();  // Invocacion
6         this.lado = lado;
7      }
8
9      public double area () {
10        return (lado*lado);
11     }
12 }
```

```
1  class Circulo extends Fig2D {
2     double radio;
3
4     public Circulo (double radio) {
5        super();  // Invocacion
6        this.radio = radio;
7     }
8
9     public double area () {
10       return (Math.PI*radio*radio);
11    }
12 }
```

En este ejemplo el nuevo constructor a invocar tan sólo tendría dos líneas de código, por lo que no nos beneficia practicamente nada en cuanto al código, más bien nos perjudica en cuanto a tiempo de ejecución; esto es así porque cada invocación a un método o constructor conlleva un retardo (paso de parámetros, etc.). Sin embargo, esta técnica puede ser muy útil en muchas ocasiones, sobre todo cuando el constructor tenga que realizar bastantes tareas, para no duplicar todo ese código en cada constructor.

Copyright ©2012- P. Pablo Garrido Abenza

5.5. Interfaces

Una *interface* es una colección de constantes y de métodos abstractos, que serán implementados por una clase concreta. Se declara como una clase, pero utilizando la palabra clave `interface`:

```
interface MiInterface {
   // Metodos abstractos
   public void metodo1();
   public void metodo2();
}
```

Una clase implementaría esta interface mediante la palabra `implements`, y esto obliga a la clase a que implemente todos los métodos declarados en la `interface`, exactamente de la misma forma pero con código. Por ejemplo, para implementar la interface anterior escribiríamos:

```
class MiClase implements MiInterface {
   // Metodos implementados de la interface MiInterface
   public void metodo1 () {
      // ...
   }
   public void metodo2 () {
      // ...
   }

   // Otros metodos
   // ...
}
```

Como se acaba de decir, una clase debe implementar todos los métodos de una interface, incluso aquellos que no se vayan a utilizar, en los que se escribirá código vacío `{}`. En caso de que alguno de los métodos no se implemente, la clase deberá declararse como abstracta, y alguna de las subclases sí que tendrá que implementar los métodos que falten.

Una clase puede implementar una o más interfaces, especificando sus nombres separados por comas. Además, también puede heredar de otra clase, que en este caso es única (no es válida la herencia múltiple). Por tanto, la sintáxis completa para declarar una clase sería:

```
class MiClase implements MiInterface1, MiInterface2 extends MiSuperclase {
   // Metodos implementados de la interface MiInterface1
   // ...
   // Metodos implementados de la interface MiInterface2
   // ...
   // Otros metodos
   // ...
}
```

Copyright © 2012 P. Pablo Garrido Abenza

Una interface, a su vez puede heredar de otras, y no sólo de una como en el caso de la herencia entre clases. En este caso, la clase que implemente esa interface deberá implementar tando los métodos que se declaren en ella como todos los métodos declarados en todas las superinterfaces.

```
interface MiInterface extends SuperInterface1, SuperInterface2, ... {
   // Metodos abstractos
   public void metodo1();
   public void metodo2();
}
```

Por defecto, todas las interfaces son abstractas, es decir, llevan implícito el modificador `abstract`. Además, de forma implícita:

- Las constantes son: publicas y estáticas (`public final static`)
- Los métodos son: abstractos y públicos (`public abstract`)

Por tanto, no será necesario precederlos con los modificadores citados. Puesto que son públicos (`public`), no podrán declararse protegidos (`protected`) ni privados (`private`), es decir, no puede restringuirse el acceso.

Copyright © 2012 P. Pablo Garrido Abenza

Interfaces vs. clases abstractas

Aparentemente, las interfaces y las clases abstractas son similares, ya que ambas declaran constantes y métodos abstractos. Sin embargo, existen varias diferencias entre ellos: (1) una clase abstracta puede declarar también métodos no abstractos, (2) una interface puede heredar de varias interfaces, mientras que una clase (abstracta o no) puede heredar de una única clase.

Usos de las interfaces

Las interfaces tienen básicamente dos **usos**:

1. Como un **contrato**: especificar que ciertas clases tengan que implementar unos métodos con unos nombres concretos y que reciban tales argumentos. Esto facilita el trabajo en equipo, donde cada desarrollador se compromete a escribir ciertas clases con cierto interface, que será el que tengan que utilizar los otros programadores, aunque sin conocer su implementación. Si sabemos que una clase implementa tal interface, ya sabemos de forma rápida qué es lo que hace, aunque la implementación en cada clase sea distinta.

2. Como **tipo de datos**: si se define un objeto cuyo tipo base es una interface (no una clase), entonces se le podrá asignar una referencia (objeto) de cualquier clase que implemente dicha interface. Y a la inversa, en cualquier lugar donde se espere un objeto de una clase, si esta clase especifica que implementa una interface, se puede pasar una referencia de un objeto que se haya declarado de esa interface como tipo. Esto se asemejaría a la herencia múltiple existente en otros lenguajes. En el presente curso, ignoraremos este uso (IGNORAR).

Algunas interfaces no tienen ningún método a implementar. Sin embargo, especificar que se implementa en una clase puede ser necesario para indicar que sobre esa clase se puede realizar cierta operación. Por ejemplo, si una clase implemente la interface `Serializable`, no tiene que implementar ningún método, pero está diciendo que los objetos de esa clase podrían almacenarse en un archivo mediante la técnica de seriación, la cual se explicará en otra unidad.

Como **ejemplo** de uso de interfaces, vamos a escribir una interface para la jerarquía de clases de figuras geométricas que vimos en la sección 5.3, correspondiente a clases abstractas. Todas las clases declaraban los métodos `area()` y `volumen()`, es decir, operaciones que se pueden realizar con figuras geométricas. Lo que se hacía era declararlos como abstractos en la clase `Figura`, y ahora los escribimos de forma separada en una interface, y especificamos que la clase `Figura` implementa dicha interface. Puesto que no la implementa se declara abstracta, igual que antes, siendo las subclases las que ya implementan estos métodos. Las subclases no necesitan especificar que implementan la interface `OperacionesFiguras`, puesto que heredan esa característica de la clase `Figura`.

```
interface OperacionesFiguras {
   // Metodos abstractos (abstract public)
   double area ();
   double volumen ();
}
```

```
import java.text.DecimalFormat;

// Clase Figura
abstract class Figura implements OperacionesFiguras {
   // Numero total de figuras 2D + 3D
   static int numFiguras = 0;
   // Formato de los resultados a mostrar
   static DecimalFormat f = new DecimalFormat ("#,##0.00");

   public String toString () {
      String s = "area=" + f.format(area()) +
                 " volumen=" + f.format(volumen());
      return (s);
   }
}
```

¿Qué conseguimos con esto? En este sencillo ejemplo, no mucho, tan sólo evitamos tener que declarar los métodos `area()` y `volumen()` en la clase `Figura`, basta con decir que implementamos la interface anterior. Pero esta interface podría ser utilizada por muchas otras clases, aunque no tengan ninguna relación entre ellas, salvo la de necesitar esas mismas operaciones, entre otras. De este modo, todas ellas podrían especificar que implementan la interface `OperacionesFiguras`, y así todas ellas nombrarían a los métodos de la misma forma; sería más fácil de recordar sus métodos que si alguien nombra los métodos como `calcularArea()`, `calculoVolumen()`,...

Copyright © 2012 P. Pablo Garrido Abenza

5.6. Enumeraciones

En Java podemos definir nuevos tipos especificando únicamente un conjunto fijo de valores constantes, de forma muy similar a la que ya ofrecía el lenguaje C mediante la palabra reservada `enum`. Al igual que, por ejemplo, el tipo `boolean` admite dos posibles valores: `false` y `true`, nosotros podemos definir nuevos tipos especificando los posibles valores que admite.

El siguiente ejemplo define un nuevo tipo llamado `DiaSemana`, que consta de 7 constantes (en mayúscula), una por cada día de la semana.

```
1 public enum DiaSemana {
2     LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
3 }
```

Como se puede observar, las 7 constantes definidas no tienen ningún valor, aunque como veremos más adelante, podrían tener uno o más valores. Estas constantes pueden utilizarse en cualquier clase de la forma `DiaSemana.LUNES`, `DiaSemana.MARTES`, ..., incluso dentro de una estructura `switch`, que en principio sólo podía utilizarse con valores de tipo `int` y `char`. En el caso de utilizar un objeto enumeración como expresión de la sentencia `switch`, no hay que especificar el nombre de la enumeración en los distintos `case`, es decir, que se debe utilizar `case LUNES`, en vez de `case DiaSemana.LUNES:`.

El siguiente programa utiliza el nuevo tipo (enumeración) `DiaSemama` que acabamos de crear; tenemos dos funciones `diaLaborable()` y `diaFestivo()` que reciben como parámetro una variable de ese nuevo tipo y averiguan si es un día laborable (lunes a viernes) o festivo (sábado a domingo), respectivamente. En realidad podría haberse definido sólo una de ellas, pues son una la inversa de la otra; cuando una retorna `true` la otra `false`, y viceversa. Sin embargo, esta forma se muestran dos posibles formas de uso de los valores de la enumeración: con una estructura `if` y con `switch`, que como se ha explicado antes, utilizan las constantes de forma distinta. Finalmente, la función principal `main()` declara una variable del nuevo tipo y la inicializa a un valor cualquiera. Dependiendo del valor asignado, las funciones retornarán `true` o `false`, y en función de ello se muestra un mensaje.

```
1  public class DiaSemanaTest {
2
3      // Método para averiguar si un dia de la semana es laborable
4      private static boolean diaLaborable (DiaSemana d) {
5          if ((d==DiaSemana.LUNES)     || (d==DiaSemana.MARTES) ||
6              (d==DiaSemana.MIERCOLES) || (d==DiaSemana.JUEVES) ||
7              (d==DiaSemana.VIERNES)) {
8              return (true);
9          } else {
10             return (false);
11         }
12     }
13
14     // Método para averiguar si un dia de la semana es festivo
15     private static boolean diaFestivo (DiaSemana d) {
```

Copyright © 2012 P. Pablo Garrido Abenza

```
16          switch (d) {
17             case LUNES:      case MARTES:
18             case MIERCOLES:  case JUEVES:
19             case VIERNES:
20                return (false);
21             case SABADO:     case DOMINGO:
22                return (true);
23             default:
24                return (false);  // No debe darse
25          }
26       }
27
28       public static void main (String args[]) {
29          DiaSemana d = DiaSemana.LUNES;
30
31          System.out.print ("Dia de la semana: " + d + " - ");
32          if ( diaLaborable(d) ) {
33          //if ( !diaFestivo(d) ) {
34             System.out.println ("laborable");
35          } else {
36             System.out.println ("no laborable");
37          }
38       }
39    }
```

El resultado del programa `DiaSemanaTest` será:

```
$ java DiaSemanaTest
Dia de la semana: LUNES - laborable
```

Copyright © 2012 P. Pablo Garrido Abenza

Con el ejemplo mostrado, vemos que los nuevos tipos `enum` se crean de forma muy similar a otros lenguajes como C. Sin embargo, esta característica de Java es más potente, porque en realidad, estos nuevos tipos son clases (heredan de forma implícita de la clase `java.lang.Enum`). Como clases que son, permiten declarar constantes, variables, constructores, y métodos en su interior. Por ejemplo, las dos funciones anteriores `diaLaborable()` y `diaFestivo()` podrían definirse dentro de la propia enumeración en vez de en cada programa que los necesite, para facilitar la reutilización de código.

```
1  public enum DiaSemana_v2 {
2     LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
3
4     // Método para averiguar si un dia de la semana es laborable
5     public boolean diaLaborable () {
6        if ((this==DiaSemana_v2.LUNES)     || (this==DiaSemana_v2.MARTES) ||
7            (this==DiaSemana_v2.MIERCOLES) || (this==DiaSemana_v2.JUEVES) ||
8            (this==DiaSemana_v2.VIERNES)) {
9            return (true);
10       } else {
11           return (false);
12       }
13    }
14
15    // Método para averiguar si un dia de la semana es festivo
16    public boolean diaFestivo () {
17       switch (this) {
18          case LUNES:      case MARTES:
19          case MIERCOLES:  case JUEVES:
20          case VIERNES:
21             return (false);
22          case SABADO:     case DOMINGO:
23             return (true);
24          default:
25             return (false);  // No debe darse
26       }
27    }
28
29    // Método para obtener el día de la semana que es
30    public int diaSemana () {
31        int cont = 1;
32        for (DiaSemana_v2 ds : DiaSemana_v2.values()) {
33           if (this.equals(ds)) return (cont);
34           cont++;
35        }
36        return (-1); // No puede darse
37    }
38 }
```

Copyright © 2012 P. Pablo Garrido Abenza

El punto y coma (;) al final de la lista de constantes de la enumeración (línea 3) sólo es necesario cuando se añaden otros elementos como variables, constructores o métodos. En el ejemplo anterior hemos añadido también un nuevo método `diaSemana()` que calcula el número del día de la semana (Lunes=1, Martes=2, ...) del objeto actual (línea 30). En éste se ha utilizado la estructura `for-each` para recorrer todos los valores que componen la enumeración como si fuera una colección.

De esta manera, el programa principal podría quedar como sigue. Destacar la forma de invocar a los métodos `diaLaborable()` y `diaFestivo()`, que ya no reciben el objeto `d` como argumento sino que se utiliza objeto invocando a un método de instancia.

```
1  public class DiaSemanaTest_v2 {
2      public static void main (String args[]) {
3          DiaSemana_v2 d = DiaSemana_v2.DOMINGO;
4
5          System.out.print ("Dia de la semana: " + d + " (" + d.diaSemana() + ") -
               ");
6          if ( d.diaLaborable() ) {
7          //if ( !d.diaFestivo() ) {
8              System.out.println ("laborable");
9          } else {
10             System.out.println ("no laborable");
11         }
12     }
13 }
```

El resultado del programa `DiaSemanaTest_v2` será:

```
$ java DiaSemanaTest_v2
Dia de la semana: DOMINGO (7) - no laborable
```

Ampliando el contenido de las enumeraciones, comentar que éstas pueden contener también **constructores**. Sin embargo, estos no se utilizan de la forma habitual de una clase: deben ser privados (`private`) o tener acceso a nivel de paquete (sin modificador `private` o `public`), y no pueden ser invocados (utilizando el operador `new`). El uso de estos constructores es asignar valores a ciertas variables constantes de clase privadas (`private final`), correspondiéndose cada una de ellas con un valor extra que se asocia a cada una de las constantes que hemos definido en la enumeración (`LUNES`, `MARTES`, ...). En otras palabras, las constantes de la lista de la enumeración pueden tener asociados uno o más valores, siempre y cuando se declaren tantas variables constantes de clase como valores especificados, y se defina un constructor que acepte también ese mismo número de valores. Es como si cada una de las constantes de la enumeración (`LUNES`, `MARTES`, ...) invocase al constructor definido en la misma enumeración pasándole los valores indicados entre paréntesis en cada una.

El siguiente ejemplo intenta ilustrar el uso de constructores. Se trata de una enumeración para los meses del año (`ENERO`, `FEBRERO`, ...), en el que asociamos a cada una de estas constantes un valor `int` con el número de días de cada mes. Caso de especificar más valores, éstos irán separados por comas dentro de los paréntesis. En este ejemplo sólo hay un valor, pero podría añadirse, por ejemplo, otros valores como la temperatura media del mes, etc.

Copyright © 2012 P. Pablo Garrido Abenza

```
1  public enum Mes {
2      // Valores de la enumeración con un valor asociado
3      ENERO (31), FEBRERO (28), MARZO (31), ABRIL (30), MAYO (31), JUNIO (30),
4      JULIO (31), AGOSTO (31), SEPTIEMBRE (30), OCTUBRE (31), NOVIEMBRE (30),
           DICIEMBRE (31);
5
6      // Variable constante de clase para que cada mes almacene su número de días
7      private final int diasMes;
8
9      // Constructor
10     private Mes (int dias) {
11        diasMes = dias;
12     }
13
14     // Método para obtener el número de días del mes (ya que diasMes es private)
15     public int getDiasMes () {
16        return (diasMes);
17     }
18 }
```

Vemos (líneas 3 y 4) que los valores de cada constante de la enumeración se especifican a continuación de cada una, entre paréntesis y separando con comas si hubiera más de un valor (en este ejemplo sólo hay un valor). Continuamos declarando una variable constante de clase (`private final`) para almacenar cada valor especificado: en este caso sólo una a la que denominamos `diasMes` (línea 7). Después declaramos un constructor privado (`private`) que recibe un valor `int` y que asigna el valor recibido a la variable constante `diasMes` (línea 10). Finalmente, para poder consultar el número de días de un objeto `Mes` es necesario el método público (`public`) `getDiasMes()`, ya que la variable constante `diasMes` es privada (línea 15).

Un programa de test de la enumeración `Mes` podría ser el siguiente, que muestra por pantalla todos los meses y su número de días, además de sumar todos los días para obtener el número de días del año (365):

```
1  public class MesTest {
2     public static void main (String args[]) {
3         int diasAño = 0;
4
5         for (Mes m : Mes.values()) {
6            // Mostramos el mes y el número de días
7            System.out.println (m + " (" + m.getDiasMes() + ")");
8            // Sumamos los días del mes
9            diasAño = diasAño + m.getDiasMes();
10        }
11        // Mostramos la suma de días del año (deberá ser 365)
12        System.out.println ("Total días año: " + diasAño);
13    }
14 }
```

Copyright © 2012 P. Pablo Garrido Abenza

El resultado de este programa será:

```
$ java MesTest
ENERO (31)
FEBRERO (28)
MARZO (31)
ABRIL (30)
MAYO (31)
JUNIO (30)
JULIO (31)
AGOSTO (31)
SEPTIEMBRE (30)
OCTUBRE (31)
NOVIEMBRE (30)
DICIEMBRE (31)
Total días año: 365
```

Copyright ©2012- P. Pablo Garrido Abenza

GLOSARIO DE ACRÓNIMOS

Siglas	Significado
ANSI	American National Standards Institute
AOT	Ahead-Of-Time (técnica de compilación)
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CISC	Complex Instruction Set Computing
DBMS	DataBase Management System
DLL	Dynamic Linked Library
ETSI	European Telecommunications Standards Institute
GNU	GNU Not Unix
GPL	GNU General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronic Engineer
ISO	International Standards Organization
JAR	Java ARchive
J2SE	Java 2 Platform, Standard Edition (~ Java SE)
J2EE	Java 2 Platform, Enterprise Edition (~ Java EE)
J2ME	Java 2 Platform, Micro Edition (~ Java ME)
JCP	Java Community Process
JDBC	Java DataBase Conectivity
JDK	Java Development Kit (~ Java 2 SDK y J2SDK)
JFC	Java Foundation Classes
JIT	Just-In-Time (técnica de compilación)
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KNI	K Native Interface (~ JNI)
ODBC	Open Database Connectivity
OSI	International Organization of Standarization
PC	Personal Computer

Copyright © 2012 P. Pablo Garrido Abenza

PDA	Personal Digital/Data Assistant
RDBMS	Relational DataBase Management System
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SDK	Standard Development Kit
SQL	Structured Query Language
SRAM	Static Random Access Memory
URL	Uniform Resource Locator
UTF-16	Unicode Transformation Format (16-bit)
WORA	Write Once, Run Anywhere

Copyright ©2012- P. Pablo Garrido Abenza