CLASE 2. CLASES EN JAVA

2.1 CONCEPTOS BÁSICOS

2.1.1 Concepto de Clase

La definición de una clase se realiza en la siguiente forma:

Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java.

Una clase es un conjunto de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos.

Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Todos los métodos se definen dentro del bloque de la clase. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos. En C la unidad fundamental son los ficheros con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave import (equivalente al #include) puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

```
[public] class NombredelaClase {
// definición de variables y métodos
...
}
```

donde la palabra public es opcional y está declarando el acceso que se va a tener a esta clase.

Cuando se crea una nueva clase en Java, se puede especificar el nivel de acceso que se quiere para las variables y los métodos definidos en la misma, eso lo pueden ver en la siguiente tabla:

public	Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instancia públicos.
protected	Sólo las subclases de la clase y nadie más puede

	acceder a las variables y métodos de instancia protegidos.
private	Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases.
friendly	Por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran friendly (amigas), lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Es lo mismo que protected.

Tabla 2.1: Nivel de Acceso

Los métodos protegidos (protected) pueden ser vistos por las clases derivadas, y también por los paquetes (packages). Todas las clases de un paquete pueden ver los métodos protegidos de ese paquete. Para evitarlo, se deben declarar como private protected, lo que hace que sólo se pueda acceder a las variables y métodos protegidos desde las clases derivadas.
Un objeto o instancia (en inglés, instance) es un representación concreta de una clase. Las clases son tipos de variables (tipos de datos), mientras que los objetos son representaciones de un tipo determinado.

```
NombredelaClase unObjeto;
NombredelaClase otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de Java deben pertenecer a una clase. No hay variables y funciones globales.

2. En un archivo se pueden definir varias clases, pero en el mismo no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.

2. Si las clases contenidas en un fichero no son public, entonces no es necesario que el fichero se llame como la clase.

2.1.2 Concepto de Interface

Una interface es un conjunto de declaraciones de funciones. Si una clase implementa (implements) una interface, debe definir todas las funciones especificadas por la interface. Las interfaces pueden definir también variables finales (constantes).

Una clase puede implementar más de una interface, representando una alternativa a la herencia múltiple.

En algunos aspectos los nombres de las interfaces pueden utilizarse en lugar de las clases.
Por ejemplo, las interfaces sirven para definir referencias a cualquier objeto de cualquiera de las clases que implementan esa interface. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste es un aspecto importante del polimorfismo.
Una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora las declaraciones de todos los métodos de las interfaces de las que deriva (a diferencia de las clases, las interfaces de Java sí tienen herencia múltiple).

2.1.3 Ejemplos de Definición de una Clase

A continuación se reproduce como ejemplo la clase **Auto** y la clase **Circulo**

```
// archivo Auto.java
public class Auto {

        public static final int numRuedas =4;
        int cantidadpuertas;
        double precio;
        boolean encendido;
        String marca;
        public Auto(int x, double y, String s) {

                cantidadpuertas = x;
                precio = y;
                marca = new String(s);
                encendido = false;

        }
        public Auto() {this(4, 0.0, "Ford"); }
        public void EncenderAuto() { encendido = true; }
        public void ApagarAuto() { encendido = false;}

} // fin de la clase Auto


//archivo Circulo.java
public class Circulo extends Geometria {

        static int numCirculos =0;
        public static final double
        PI=2.14159265358979323846;
        public double x, y, r;
        public Circulo(double x, double y, double r) {
```

```
            this.x=x; this.y=y; this.r=r;
            numCirculos++;

        }
        public Circulo(double r) { this(0.0, 0.0, r); }
        public Circulo(Circulo c) { this(c.x, c.y, c.r); }
        public Circulo() { this(0.0, 0.0, 1.0); }
        public double perimetro() { return 2.0 * PI * r; }
        public double area() { return PI *r *r;}

        // método de objeto para comparar círculos
        public Circulo elMayor(Circulo c) {
        if (this.r>=c.r) return this;
        else return c;
        }.

        // método de clase para comparar círculos
        public static Circulo elMayor(Circulo c, Circulo d) {
        if (c.r>=d.r) return c;
        else return d;
        }

} // fin de la clase Circulo
```

En estos ejemplos pueden ver cómo se definen las variables miembro y los métodos dentro de la clase. Dichas variables y métodos pueden ser de objeto o de clase (static). Además observen que el nombre del fichero coincide con el de la clase public con la extensión *.java

2.2 VARIABLES MIEMBRO

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está centrada en los datos. Una clase está constituida por datos y métodos que operan sobre esos datos.

2.2.1 Variables miembro de objeto

Cada objeto, es decir cada representación concreta de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas campos) pueden ser de tipos primitivos (boolean, int, long, double, …) o referencias a objetos de otra clase.

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de tipos primitivos se inicializan siempre de modo automático, incluso antes de llamar al constructor (false para boolean, el carácter nulo para char y cero para los tipos numéricos).

De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también inicializarse explícitamente en la declaración, como las variables locales, por medio de constantes o llamadas a métodos. Por ejemplo,
long nDatos = 100;
Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada objeto que se crea de una clase tiene su propia copia de las variables miembro. Por
ejemplo, cada objeto de la clase Auto tiene sus propio precio, marca, modelo, cantidad de puertas y estado del auto.

Los métodos de objeto se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama argumento implícito. Por ejemplo, para poner el auto en encendido de un objeto de la clase Auto llamado c1 se escribirá: c1.EncenderAuto();.

Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra this y el operador punto.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: public, private, protected y package (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (public y package), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

2.2.2 Variables miembro de clase (static)

Una clase puede tener variables propias y no de cada objeto. A estas variables se les llama variables de clase o variables static. Las variables static se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo numRuedas en la clase Auto) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados).

Las variables de clase se crean anteponiendo la palabra static a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro.
Por ejemplo, Auto.numRuedas es una variable de clase. Si no se les da valor en la declaración, las variables miembro static se inicializan con los valores por defecto para los tipos primitivos (false para boolean, el carácter nulo para char y cero para los tipos numéricos), y con null si es una referencia.
Las variables miembro static se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método static o en cuanto se utiliza una variable static de dicha clase. Lo importante es que las variables miembro static se inicializan siempre antes que cualquier objeto de la clase.

2.3 VARIABLES FINALES

Una variable de un tipo primitivo declarada como final no puede cambiar su valor a lo largo de la ejecución del programa y es una constante.

Java permite separar la definición de la inicialización de una variable final. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos.

La variable final así definida es constante (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados final.

Declarar como final un objeto miembro de una clase hace constante la referencia, pero no el propio objeto, que puede ser modificado a través de otra referencia. En Java no es posible hacer que un objeto sea constante.

2.4 MÉTODOS (FUNCIONES MIEMBRO)
2.4.1 Métodos de objeto

Los métodos son funciones definidas dentro de una clase. Salvo los métodos static o de clase, se aplican siempre a un objeto de la clase por medio del operador punto (.). Dicho objeto es su argumento implícito. Los métodos pueden además tener otros argumentos explícitos que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama declaración o header; el código comprendido entre las llaves {…} es el cuerpo o body del método. Considérese el siguiente método tomado de la clase Circulo:

```
public Circulo elMayor(Circulo c) { // header y comienzo del
método
    if (this.r>=c.r) return this;          // body
    else return c;                          // body
}                                          // final del método
```

El header consta del calificador de acceso (public, en este caso), del tipo del valor de retorno (Circulo en este ejemplo, void si no tiene), del nombre de la función y de una lista de argumentos explícitos entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen visibilidad directa de las variables miembro del objeto que es su argumento implícito, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia this, de modo discrecional (como en el ejemplo anterior con this.r) o si alguna variable local o argumento las

oculta.

El valor de retorno puede ser un valor de un tipo primitivo o una referencia. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de interface. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir variables locales. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

2.4.2 Métodos sobrecargados (overloaded)

Java permite métodos sobrecargados (overloaded), es decir, métodos distintos que tienen el mismo nombre, pero que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase Circulo presenta dos casos de métodos sobrecargados: los cuatro constructores y los dos métodos llamados elMayor().

A la hora de llamar a un método sobrecargado, Java sigue unas reglas para determinar el método concreto que debe llamar:

> **1.** Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
>
> **2.** Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo char a int, int a long, float a double, etc.) y se llama el método correspondiente.
>
> **2.** Si sólo existen métodos con argumentos de un tipo más restringido (por ejemplo, int en vez de long), el programador debe hacer un cast explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
>
> **4.** El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la sobrecarga de métodos es la redefinición. Una clase puede redefinir (override) un método heredado de una superclase. Redefinir un

método es dar una nueva definición.

En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la herencia.

2.4.3 Paso de argumentos a métodos

En Java los argumentos de los tipos primitivos se pasan siempre por valor. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las referencias se pasan también por valor, pero a través de ellas se pueden modificar los objetos referenciados.

En Java no se pueden pasar métodos como argumentos a otros métodos. Lo que se puede hacer en Java es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear variables locales de los tipos primitivos o referencias.

Estas variables locales dejan de existir al terminar la ejecución del método. Los argumentos formales de un método (las variables que aparecen en el header del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

Si un método devuelve this (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

String numeroComoString = "8.978";
float p = Float.valueOf(numeroComoString).floatValue();
donde el método valueOf(String) de la clase java.lang.Float devuelve un objeto de la clase Float sobre el que se aplica el método floatValue(), que finalmente devuelve una variable primitiva de tipo float. El ejemplo anterior se podía desdoblar en las siguientes sentencias:
String numeroComoString = "8.978";
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (**.**) que, como todos los operadores de Java excepto los de asignación, se ejecuta de izquierda a derecha.

2.4.4 Métodos de clase (static)

Análogamente, puede también haber métodos que no actúen sobre objetos

concretos a través del operador punto. A estos métodos se les llama métodos de clase o static. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia this. Un ejemplo típico de métodos static son los métodos matemáticos de la clase java.lang.Math (sin(), cos(), exp(), pow(), etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra static. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, Math.sin(ang), para calcular el seno de un ángulo).

2.4.5 Constructores

Un punto clave de la Programación Orientada Objetos es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. Java no permite que haya variables miembro que no estén inicializadas. Ya se ha dicho que Java inicializa siempre con valores por defecto las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de constructores.

Un constructor es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembro de la clase.

Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase. Su argumento implícito es el objeto que se está creando.

De ordinario una clase tiene varios constructores, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos sobrecargados). Se llama constructor por defecto al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un constructor de una clase puede llamar a otro constructor previamente definido en la misma clase por medio de la palabra this. En este contexto, la palabra this sólo puede aparecer en la primera sentencia de un constructor.

El constructor de una sub-clase puede llamar al constructor de su super-clase por medio de la palabra super, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.
El constructor es tan importante que, si el programador no prepara ningún constructor para una clase, el compilador crea un constructor por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, y los Strings y las demás referencias a objetos a null. Si hace falta, se llama al constructor de la super-clase para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los constructores pueden tener también los modificadores de acceso public, private, protected y package. Si un constructor es private, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos public y static (factory methods) que llamen al constructor y devuelvan un objeto de esa clase.

Dentro de una clase, los constructores sólo pueden ser llamados por otros constructores o por métodos static. No pueden ser llamados por los métodos de objeto de la clase.

Un ejemplo del uso de los constructores se puede ver a continuación:

```
Circulo c1,c2,c3;                          //solo hemos declarado
las variables
c1 = new Circulo();
c2 = new Circulo(1.1,2.2,0.0);
c3 = new Circulo(1.1);
```

2.4.6 Inicializadores

Por motivos que se verán más adelante, Java todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los inicializadores, que pueden ser static (para la clase) o de objeto.

2.4.6.1 Inicializadores static

Un inicializador static es un algo parecido a un método (un bloque {…} de código, sin nombre y sin argumentos, precedido por la palabra static) que se llama automáticamente al crear la clase (al utilizarla por primera vez). También se diferencia del constructor en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados.

Los inicializadores static se crean dentro de la clase, como métodos sin nombre, sin argumentos y sin valor de retorno, con tan sólo la palabra static y el código entre llaves {...}. En una clase pueden definirse varios inicializadores static, que se llamarán en el orden en que han sido definidos.

Los inicializadores static se pueden utilizar para dar valor a las variables static.

2.4.6.2 Inicializadores de objeto

A partir de Java 1.1 existen también inicializadores de objeto, que no llevan la palabra static. Se utilizan para las clases anónimas, que por no tener nombre no pueden tener constructor. En este caso, los inicializadores de objeto se llaman cada vez que se crea un objeto de la clase anónima.

2.4.7 Resumen del proceso de creación de un objeto

El proceso de creación de objetos de una clase es el siguiente:

1. Al crear el primer objeto de la clase o al utilizar el primer método o variable static se localiza la clase y se carga en memoria.

2. Se ejecutan los inicializadores static (sólo una vez).

2. Cada vez que se quiere crear un nuevo objeto:

a) se comienza reservando la memoria necesaria
b) se da valor por defecto a las variables miembro de los tipos primitivos
c) se ejecutan los inicializadores de objeto
d) se ejecutan los constructores

2.4.8 ALCANCE DE OBJETOS Y RECICLADO DE MEMORIA

Los objetos tienen un tiempo de vida y consumen recursos durante el mismo. Cuando un objeto no se va a utilizar más, debería liberar el espacio que ocupaba en la memoria de forma que las aplicaciones no la agoten (especialmente las grandes).
En Java, la recolección y liberación de memoria es responsabilidad de un thread (es un programa con características especiales que se explica en la clase 7) llamado **automatic garbage collector** (recolector automático de basura). Este thread monitoriza el alcance de los objetos y marca los objetos que se han salido de alcance. Veamos un ejemplo:

```
String s;   // no se ha asignado todavía
s = new String( "abc" ); // memoria asignada
s = "def";  // se ha asignado nueva memoria (nuevo objeto)
```

Más adelante veremos en detalle la clase String, pero una breve descripción de lo que hace esto es crear un objeto String y rellenarlo con los caracteres "abc" y crear otro (nuevo) String y colocarle los caracteres "def".
En esencia se crean dos objetos:

Objeto String "abc"
Objeto String "def"

Al final de la tercera sentencia, el primer objeto creado, de nombre **s**, que contiene "abc" se ha salido de alcance. No hay forma de acceder a él. Ahora se tiene un nuevo objeto llamado s que contiene "def". Es marcado y eliminado en la siguiente iteración del thread reciclador de memoria.

2.5 PACKAGES

2.5.1 ¿Qué es un package?

Un package es una agrupación de clases. Además, el usuario puede crear sus propios packages. Para que una clase pase a formar parte de un package llamado pkgName, hay que introducir en ella la sentencia:
package pkgName;
que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los packages se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un package puede constar de varios nombres unidos por puntos (los propios packages de Java siguen esta norma, como por ejemplo java.awt.event).

Todas las clases que forman parte de un package deben estar en el mismo directorio. Los
nombres compuestos de los packages están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los nombres de las clases de Java sean únicos en Internet.

Es el nombre del package lo que permite obtener esta característica. Una forma de conseguirlo es incluir el nombre del dominio (quitando quizás el país), como por ejemplo en el package siguiente:

es.ceit.jgjalon.infor2.ordenar

Las clases de un package se almacenan en un directorio con el mismo nombre largo (path) que el package. Por ejemplo, la clase,

es.ceit.jgjalon.infor2.ordenar.QuickSort.class

debería estar en el directorio,

CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de Java (clases del sistema o de usuario), en este caso la posición del directorio es en los discos locales del ordenador. Los packages se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.

2. Para evitar conflictos de nombres. En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del package.

2. Para ayudar en el control de la accesibilidad de clases y miembros.

2.5.2 ¿Cómo funcionan los packages?

Con la sentencia import packname; se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de clases, Java da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del package.

El importar un package no hace que se carguen todas las clases del package: sólo se cargarán las clases public que se vayan a utilizar. Al importar un package no se importan los sub-packages.

Éstos deben ser importados explícitamente, pues en realidad son packages distintos. Por ejemplo, al importar java.awt no se importa java.awt.event.

Es posible guardar en jerarquías de directorios diferentes los ficheros *.class y *.java, con objeto por ejemplo de no mostrar la situación del código fuente. Los packages hacen referencia a los ficheros compilados *.class.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del package más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia import permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del package importado. Se importan por defecto el package java.lang y el package actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar import: para una clase y para todo un package:
import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.*;
que deberían estar en el directorio:
classpath\es\ceit\jgjalon\infor2\ordenar
El cómo afectan los packages a los permisos de acceso de una clase se estudiaran más adelante.

2.6 HERENCIA

2.6.1 Concepto de herencia

Se puede construir una clase a partir de otra mediante el mecanismo de la herencia. Para indicar que una clase deriva de otra se utiliza la palabra extends, como por ejemplo:
class CirculoGrafico extends Circulo {...}
Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser redefinidas (overridden) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma, lo pueden ver, como si la sub-clase (la clase derivada) "contuviera" un objeto de la super-clase; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de Java creadas por el programador tienen una super-clase. Cuando no se indica explícitamente una super-clase con la palabra extends, la clase deriva de java.lang.Object, que es la clase raíz de toda la jerarquía de clases de Java. Como consecuencia, todas las clases tienen algunos métodos que han heredado de Object.

La composición (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la herencia en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace private).

2.6.2 Redefinición de métodos heredados

Una clase puede redefinir (volver a definir) cualquiera de los métodos heredados de su super-clase que no sean final. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Las métodos de la super-clase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra super desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden ampliar los derechos de acceso de la super-clase (por ejemplo ser public, en vez de protected o package), pero nunca restringirlos.

Los métodos de clase o static no pueden ser redefinidos en las clases derivadas.

2.6.3 this y super

Al acceder a variables de instancia de una clase, la palabra clave **this** hace referencia a los miembros de la propia clase. Veamos un ejemplo del uso de **this**:

```
//archivo NuestraClase.java

public class NuestraClase {

        int i;

        public NuestraClase() { i = 10; } // Este constructor
        establece el valor de i

        public NuestraClase( int valor ) {

                this.i = valor; // i = valor

        }

        public void Suma_a_i( int j ) {

                i = i + j;

                }

        }
```

Aquí this.i se refiere al entero i en la clase NuestraClase.
Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra clave **super**:

```
//archivo NuestraNuevaClase.java

public class NuestraNuevaClase extends NuestraClase {

        public void Suma_a_i( int j ) {

                i = i + ( j/2 );

                super.Suma_a_i( j );

        }

}
```

En el siguiente código, el constructor establecerá el valor de i a 10, después lo cambiará a 15 y finalmente el método Suma_a_i() de la clase padre (NuestraClase) lo dejará en 25:

```
NuestraNuevaClase mnc;
mnc = new NuestraNuevaClase();
mnc.Suma_a_i( 10 );
```

2.6.4 Constructores en clases derivadas

Ya se comentó que un constructor de una clase puede llamar por medio de la palabra this a otro constructor previamente definido en la misma clase. En este contexto, la palabra this sólo puede aparecer en la primera sentencia de un constructor.

De forma análoga el constructor de una clase derivada puede llamar al constructor de su super-clase por medio de la palabra super(), seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la super-clase. De esta forma, un constructor sólo tiene que inicializar directamente las variables no heredadas.

La llamada al constructor de la super-clase debe ser la primera sentencia del constructor, excepto si se llama a otro constructor de la misma clase con this(). Si el programador no la incluye, Java incluye automáticamente una llamada al constructor por defecto de la super-clase, super(). Esta llamada en cadena a los constructores de las super-clases llega hasta el origen de la jerarquía de clases, esto es al constructor de Object.

Como ya se ha dicho, si el programador no prepara un constructor por defecto, el compilador crea uno, inicializando las variables de los tipos primitivos a sus valores por defecto, y los Strings y demás referencias a objetos a null. Antes, incluirá una llamada al constructor de la super-clase.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con new, de la que se encarga el garbage collector), es importante llamar a los finalizadores de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el finalizador de la sub-clase deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la forma super.finalize(). Los métodos finalize() deben ser al menos protected, ya que el método finalize() de Object lo es, y no está permitido reducir los permisos de acceso en la herencia.

2.6.5 UN CASO DE EJEMPLO

A continuación escribiremos dos clases:

```
//Archivo Vehiculo

class Vehiculo {

        String marca, modelo;
```

```
        Vehiculo() {
                marca = new String();
                modelo = new String();
        }
        Vehiculo (String a, String b) {
                marca = new String(a);
                modelo = new String(b);
        }
}

//Archivo Auto
class Auto extends Vehículo {
        int cantidadPuertas;
        Auto() {
                super();
                cantidadPuertas=0;
        }
        Auto (String a, String b, int c) {
                super(a, b);
                cantidadPuertas=c;
        }
}
```

Como pueden ver, hemos creado dos clases: Vehículo y Auto. Entre ellas hay una relación de herencia (Auto hereda a Vehículo).
Sobre lo que quiero llamarles la atención es sobre el uso de super en los constructores de la clase hija (Auto), allí la primera sentencia debe ser la invocación al constructor de la clase padre, no puede haber ninguna sentencia antes de esto) O sea, si cambiamos el constructor sin parámetros de la clase Auto por:

```
Auto() {
        cantidadPuertas=0;
        super();
}
```

esto provocará un error al compilarlo.

2.6.6 Clases y métodos abstractos

Una clase abstracta (abstract) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra abstract, como por ejemplo,

public abstract class Geometria { ... }

Una clase abstract puede tener métodos declarados como abstract, en cuyo caso no se da definición del método. Si una clase tiene algún método abstract es obligatorio que la clase sea abstract. En cualquier sub-clase este método deberá bien ser redefinido, bien volver a declararse como abstract (el método y la sub-clase).

Una clase abstract puede tener métodos que no son abstract. Aunque no se puedan crear objetos de esta clase, sus sub-clases heredarán el método completamente a punto para ser utilizado.

Como los métodos static no pueden ser redefinidos, un método abstract no puede ser static.

2.6.7 POLIMORFISMO

El polimorfismo tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama vinculación (binding). La vinculación puede ser temprana (en tiempo de compilación) o tardía (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en Java se utiliza siempre vinculación tardía, excepto si el método es final. El polimorfismo es la opción por defecto en Java.
La vinculación tardía hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea el tipo de objeto y no el tipo de la referencia lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el polimorfismo necesita evaluación tardía.

El polimorfismo permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.
El polimorfismo puede hacerse con referencias de super-clases abstract, super-clases normales e interfaces. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las interfaces permiten ampliar muchísimo las posibilidades del polimorfismo.

2.7 CLASES Y MÉTODOS FINALES

Recuérdese que las variables declaradas como final no pueden cambiar su valor una vez que han sido inicializadas. En este apartado se van a presentar otros dos usos de la palabra final.

Una clase declarada final no puede tener clases derivadas. Esto se puede hacer por motivos de seguridad y también por motivos de eficiencia, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un método declarado como final no puede ser redefinido por una clase que derive de su propia clase.

2.8 INTERFACES

2.8.1 Concepto de interface

Una interface es un conjunto de declaraciones de métodos (sin definición). También puede definir constantes, que son implícitamente public, static y final, y deben siempre inicializarse en la declaración. Estos métodos definen un tipo de conducta. Todas las clases que implementan una determinada interface están obligadas a proporcionar una definición de los métodos de la interface, y en ese sentido adquieren una conducta o modo de funcionamiento.

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las interfaces, separados por comas, detrás de la palabra implements, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo
implements Dibujable {
...
}
```

¿Qué diferencia hay entre una interface y una clase abstract? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase abstract puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas diferencias importantes:

1. Una clase no puede heredar de dos clases abstract, pero sí puede heredar de una clase abstract e implementar una interface,

o bien implementar dos o más interfaces.

2. Una clase no puede heredar métodos -definidos- de una interface, aunque sí constantes.

2. Las interfaces permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, inpedientemente de su situación en la jerarquía de clases de Java.

4. Las interfaces permiten "publicar" el comportamiento de una clase desvelando un mínimo de información.

5. Las interfaces tienen una jerarquía propia, independiente y más flexible que la de las clases, ya que tienen permitida la herencia múltiple.

6. De cara al polimorfismo, las referencias de un tipo interface se pueden utilizar de modo similar a las clases abstract.

2.8.2 Definición de interfaces

Una interface se define de un modo muy similar a las clases. Por ejemplo:

```
// fichero Dibujable.java
import java.awt.Graphics;
public interface Dibujable {
public void setPosicion(double x, double y);
public void dibujar(Graphics dw);
}
```

Cada interface public debe ser definida en un fichero *.java con el mismo nombre de la interface. Los nombres de las interfaces suelen comenzar también con mayúscula.

Las interfaces no admiten más que los modificadores de acceso public y package. Si la interface no es public no será accesible desde fuera del package (tendrá la accesibilidad por defecto, que es package). Los métodos declarados en una interface son siempre public y abstract, de modo implícito.

2.8.3 Herencia en interfaces

Entre las interfaces existe una jerarquía (independiente de la de las clases) que permite herencia simple y múltiple. Cuando una interface deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una interface puede derivar de varias interfaces. Para la herencia de interfaces se utiliza asimismo la palabra extends, seguida por el nombre de las interfaces de las que deriva, separadas por comas.

Una interface puede ocultar una constante definida en una super-interface

definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una super-interface.

Las interfaces no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha interface dejarán de funcionar, a menos que implementen el nuevo método.

2.8.4 Utilización de interfaces

Las constantes definidas en una interface se pueden utilizar en cualquier clase (aunque no implemente la interface) precediéndolas del nombre de la interface, como por ejemplo (suponiendo que PI hubiera sido definida en Dibujable):
area = 2.0*Dibujable.PI*r;
Sin embargo, en las clases que implementan la interface las constantes se pueden utilizar directamente, como si fueran constantes de la clase. De cara al polimorfismo, el nombre de una interface se puede utilizar como un nuevo tipo de referencia. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la interface.

Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

2.9 PERMISOS DE ACCESO EN JAVA

Una de las características de la Programación Orientada a Objetos es el encapsulamiento que fue visto en el capítulo 1. Los permisos de acceso de Java son una de las herramientas para conseguir esta finalidad.

2.9.1 Accesibilidad de los packages

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un package es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos packages que se encuentren en la variable CLASSPATH del sistema.

2.9.2 Accesibilidad de clases o interfaces

En principio, cualquier clase o interface de un package es accesible para todas las demás clases del package, tanto si es public como si no lo es. Una clase public es accesible para cualquier otra clase siempre que su package sea accesible. Recuérdese que las clases e interfaces sólo pueden ser public o package (la opción por defecto cuando no se pone ningún modificador).

2.9.3 Accesibilidad de las variables y métodos miembros de una clase

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia this) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.

2. Los miembros private de una clase sólo son accesibles para la propia clase.

2. Si el constructor de una clase es private, sólo un método static de la propia clase puede crear objetos.

Desde una sub-clase:

1. Las sub-clases heredan los miembros private de su super-clase, pero sólo pueden acceder a ellos a través de métodos public, protected o package de la super-clase.

Desde otras clases del package:

1. Desde una clase de un package se tiene acceso a todos los miembros que no sean private de las demás clases del package.

Desde otras clases fuera del package:

1. Los métodos y variables son accesibles si la clase es public y el miembro es public.

2. También son accesibles si la clase que accede es una sub-clase y el miembro es protected.

4.1 TRANSFORMACIONES DE TIPO: CASTING

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de int a double, o de float a long. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

4.1.1 Conversión de tipos primitivos

La conversión entre tipos primitivos es más sencilla. En Java se realizan de modo automático conversiones implícitas de un tipo a otro de más precisión, por ejemplo de int a long, de float a double, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar

sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto (más amplio) que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son conversiones inseguras que pueden dar lugar a errores (por ejemplo, para pasar a short un número almacenado como int, hay que estar seguro de que puede ser representado con el número de cifras binarias de short). A estas conversiones explícitas de tipo se les llama cast. El cast se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;

result = (long) (a/(b+c));
```

En Java no se puede convertir un tipo numérico a boolean.