Tema 5: Clases en Java

Las clases son el centro de la Programación Orientada a Objetos (OOP - Object Oriented Programming). Algunos de los conceptos más importantes de la POO son los siguientes:

- **Encapsulación**. Las clases pueden ser declaradas como públicas (public) y como package (accesibles sólo para otras clases del package). Las variables miembro y los métodos pueden ser public, private, protected y package. De esta forma se puede controlar el acceso y evitar un uso inadecuado.

- **Herencia**. Una clase puede derivar de otra (extends), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede añadir nuevas variables y métodos y/o redefinir las variables y métodos heredados.

- **Polimorfismo**. Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código.

En este Capítulo se presentan las clases tal como están implementadas en el lenguaje Java.

5.1. Concepto de Clase.-

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

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

donde la palabra public es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del package. Todos los métodos y variables deben ser definidos dentro del bloque {...} de la clase.

Un objeto (en inglés, instance) es un ejemplar concreto de una clase. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

```
Classname unObjeto;
Classname 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. Si una clase deriva de otra (extends), hereda todas sus variables y métodos.

3. Java tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los

usuarios.

4. Una clase sólo puede heredar de una única clase (en Java no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object. La clase Object es la base de toda la jerarquía de clases de Java.

5. En un fichero se pueden definir varias clases, pero en un fichero 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.

6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.

7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia this.

8. Las clases se pueden agrupar en packages, introduciendo una línea al comienzo del fichero (package packageName;). Esta agrupación en packages está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

5.2 Ejemplo de una clase.-

A continuación se reproduce como ejemplo la clase Circulo

```
// fichero Circulo.java

public class Circulo
{
     public static final double PI=3.14159265358979323846;
     public double x, y, r;


     public Circulo(double x, double y, double r)
     {
          this.x=x;
          this.y=y;
          this.r=r;
     }
     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)
    {
        Ciruclo mayor = c;
        if (this.r>=c.r)
            mayor =  this;
        return mayor;
    }

} // fin de la clase Circulo
```

En este ejemplo se ve cómo se definen las variables miembro y los métodos (cuyos nombres se han resaltado en negrita) dentro de la clase. Dichas variables y métodos pueden ser de objeto o de clase (static). Se puede ver también cómo el nombre del fichero coincide con el de la clase public con la extensión *.java.

5.3 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 unos datos y unos métodos que operan sobre esos datos.

5.3.1 Variables miembro de objeto.-

Cada objeto, es decir cada ejemplar concreto 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 (composición).

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 Circulo tiene sus propias coordenadas del centro x e y, y su propio valor del radio r.

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 calcular el área de un objeto de la clase Circulo llamado c1 se escribirá:

c1.area();.

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.

5.3.2 Variables miembro de clase (static).-

Una clase puede tener variables propias de la clase 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 PI en la clase Circulo) o variables que sólo tienen sentido para toda la clase.

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.

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.

5.4 Variables finales.-

Una variable de un tipo primitivo declarada como final no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como 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.

5.5 Métodos (funciones miembro).-

5.5.1 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) // body
return this; // body
else // body
return c; // body
} // final del método
```

```
// método de objeto para comparar círculos
public Circulo elMayor(Circulo c) // header
{ //comienzo del método
     Ciruclo mayor = c; // body
     if (this.r>=c.r)
          mayor =  this; // body
     return mayor; // body
} // final del método
```

El header consta del cualificador 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).

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.

5.5.2 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 casos de métodos sobrecargados: los cuatro constructores.

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.

3. 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.

5.5.3 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 método1. 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 .

3.5.4 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).

3.5.5 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 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.

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.

5.6 Packages.-

5.6.1 Qué es un package.-

Un package es una agrupación de clases. En la API de Java 1.1 había 22 packages; en Java 1.2 hay 59 packages, lo que da una idea del "crecimiento" experimentado por el lenguaje.

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:

- Para agrupar clases relacionadas.
- Para evitar conflictos de nombres (se recuerda que el dominio de nombres de Java es la Internet). 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.
- Para ayudar en el control de la accesibilidad de clases y miembros.

5.6.2 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
```

EJERCICIOS RESUELTOS.-

- *Escribir un programa que permita convertir horas en notación de 24 horas a su equivalente en notación de 12 horas. Por ejemplo, `23:50` debe ser convertida en `11:50 PM`.*

  En primer lugar intentamos crear la clase *Reloj* que representará en reloj en formato de 24 horas y tendrá los atributos y operaciones necesarios para mantener una hora en dicho formato. Su código es el siguiente:

```
// Reloj.java

class Reloj
{
      // Estado de la clase
      private int horas, minutos;

      // Metodos de la clase

      // Constructores
      public Reloj ()
      {
            horas = 0;
            minutos = 0;
      }

      public Reloj (int horas, int minutos)
      {
            this.setHoras (horas);
            this.setMinutos (minutos);
      }

      // Selectores
      public void setHoras (int horas)
      {
            this.horas = horas;
            this.comprobar ();
      }

      public void setMinutos (int minutos)
```

```
	{
		this.minutos = minutos;
		this.comprobar ();
	}

	public int getHoras ()
	{
		return this.horas;
	}

	public int getMinutos ()
	{
		return minutos;
	}

	// Metodos propios
	private void comprobar ()
	{
		while (this.minutos > 59)
		{
			this.minutos -= 60;
			this.horas++;
		}

		while (horas > 23)
			horas -= 24;
	}
}// Reloj
```

A continuación basándonos en esta clase se creará la clase *Reloj12* para dar soporte a las horas en formato de 12. Su código es el siguiente:

```
// Reloj12.java

import Reloj;

class Reloj12 extends Reloj
{
```

```
// Estado de la clase
String indicador;

// Metodos de la clase
// Constructores
Reloj12 ()
{
      super (); // llamada al constructor base
      indicador = new String ("AM");

}

Reloj12 (int horas, int minutos)
{
      super (horas, minutos);
      this.setIndicador ();
}

Reloj12 (int horas, int minutos, String indicador)
{
      super (horas,minutos);
      this.setIndicador (); // ajustar las horas a formato 12
      this.setIndicador (indicador);
}


// Selectores
public void setHoras (int horas)
{
      super.setHoras (horas);
      this.setIndicador ();
}

public void setMinutos (int minutos)
{
      super.setMinutos (minutos);
      this.setIndicador ();
}
```

```
	private void setIndicador ()
	{
		while (super.getMinutos () > 59)
		{
			super.setMinutos (super.getMinutos () - 60);//minutos -= 60
			super.setHoras (super.getHoras () +1 ); // horas++;
		}


		while ( super.getHoras () > 11)
		{
			this.setIndicador ("PM");
			super.setHoras (super.getHoras () - 12); // horas -= 12
		}
	}

	public void setIndicador (String indicador)
	{
		if (indicador.equals ("AM")|| indicador.equals ("am"))
			this.indicador = indicador;
		else
			if (indicador.equals ("PM") || indicador.equals ("pm"))
				this.indicador = indicador ;
	}

	public String getIndicador ()
	{
		return indicador;
	}

} // Reloj12
```

Finalmente nos plantemos resolver el problema propuesto utilizando las clases anteriores. Para ello creamos la clase aplicación *HorasApp*, que tendrá un método *main ()* encargado de resolver el problema. Su código es el siguiente:

```
// HorasApp.java
import Reloj12;
```

```
class HorasApp
{

      public static void main (String args[])
      {
            int h,m;
            int horas,minutos;
            Reloj12 reloj ;

            if (args.length != 2) // comprobar el numero de parametros
            {
                  System.out.println ("Argumentos no validos");
            }
            else
            {
                  h= Integer.valueOf (args[0]).intValue(); // convertir a int
                  m = Integer.valueOf(args[1]).intValue (); // convertir a int

                  reloj = new Reloj12 (h, m); // crear el relor de 12 horas


                  // Sacar resultados
                  System.out.println ("La hora en formato 12 horas es:");
                  System.out.println (reloj.getHoras()+":"+reloj.getMinutos()
                                          + " " + reloj.getIndicador ());
            }
      }
}// HorasApp
```

- *Escribir un programa que lea una lista de 10 números de la entrada estándar y calcule:*

  a) La suma de todos los números que sean pares.

  b) Su media.

  c) La lista de números ordenados.

En primer lugar nos planteamos el crear un clase que contenga la lista de los 10 números y sus operaciones propias. Dicha clase la llamaremos *Array*. Su código es el siguiente:

```
// Array.java

class Array
{

	// Estado de la clase.
	int array[];

	// Metodos de la clase
	// Constructores
	Array ()
	{
		array = null; // no hay tamaño definido
	}

	Array (int dimension)
	{
		array = new int[dimension];
	}

	Array (Array from)
	{
		array = new int[from.getLongitud()];
		this.assign (from);
	}

	// Selectores

	void setElemento (int n, int valor)
	{
		array[n] = valor;
	}

	int getElemento (int n)
	{
		return array[n];
	}

	int getLongitud ()
```

```
{
	return array.length;
}
// Metodos propios
void assign (Array from)
{
	for (int i = 0; i < from.array.length; i++)
		array[i] = from.array[i];
}

Array ordenar ()
{
	Array ordenado = new Array (this.getLongitud());
	int i,j,aux;

	ordenado.assign (this);

	for (i = 0; i < ordenado.array.length -1 ; i++)
		for (j = i ; j < ordenado.array.length; j++)
			{
				if (ordenado.array[i] > ordenado.array[j])
				{
					aux = ordenado.array[i];
					ordenado.array[i] = ordenado.array[j];
					ordenado.array[j] = aux;
				}
			}

	return ordenado;
}

float getMedia ()
{

	return getSuma() /array.length ;
}

boolean espar (int n)
{
```

```
            return ((array[n] % 2) == 0);
      }

      int getSuma ()
      {
            int suma = 0;

            for (int i = 0; i < array.length; i++)
                  suma += array[i];

            return suma;
      }

}//Array
```

A continuación nos plantemos resolver el problema planteado utilizando la clase *Array*. Para ello creamos la clase *ArrayApp*, cuyo código es el siguiente:

```
// ArrayApp.java

import java.io.*; // clases de entrada-salida
import Array;

class ArrayApp
{
      // definicion de constantes
      static final int N = 10; // Numero de elementos

      public static void main (String args[])
      {
            ArrayApp programa= new ArrayApp();
            Array miarray = new Array (N);

            programa.leerArray (miarray);
            programa.sacarmedia (miarray);
            programa.sacarorden (miarray);
            programa.sacarsumapar(miarray);
```

```
	}

	public void sacarmedia (Array a)
	{
		Pantalla.escribirString ("La media es: "+ a.getMedia());
	}

	public void sacarorden (Array a)
	{

		Array orden = a.ordenar ();

		Pantalla.escribirString ("El array ordenado es:");

		for (int i = 0; i < orden.getLongitud(); i++)
			Pantalla.escribirString (" " + orden.getElemento(i));
	}

	public void sacarsumapar (Array a)
	{

		Array par = new Array (N);

		for (int i = 0; i < par.getLongitud(); i++)
		{
			if (a.espar(i))
				par.setElemento (i, a.getElemento(i));
		}
		Pantalla.escribirString ("La suma de los pares es :" + par.getSuma());
	}
	public void leerArray (Array a) throws IOException // propagar excep.
	{
		for (int i = 0; i < a.getLongitud (); i++)
		{
			Patalla.escribirString("> ");
			a.setElemento(i,Integer.valueOf(Teclado.leerInt());
		}
    }
}// ArrayApp
```