

# 1.

## Programación Orientada a Objetos

### Conceptos de la POO

La programación orientada a objetos establece un equilibrio entre la importancia de los procesos y los datos, mostrando un enfoque más cercano al pensamiento del ser humano. Se introduce un aspecto novedoso respecto al anterior paradigma: la herencia, facilitando el crecimiento y la mantenibilidad.

Las bases de la programación orientada a objetos son: abstracción, encapsulación, modularidad y jerarquización.

La **abstracción** es un proceso mental de extracción de las características esenciales, ignorando los detalles superfluos. Resulta ser muy subjetiva dependiendo del interés del observador, permitiendo abstracciones muy diferentes de la misma realidad

La **encapsulación** es ocultar los detalles que dan soporte a un conjunto de características esenciales de una abstracción. Existirán dos partes, una visible que todos tienen acceso y se aporta la funcionalidad, y una oculta que implementa los detalles internos.

La **modularidad** es descomponer un sistema en un conjunto de partes. Aparecen dos conceptos muy importantes: acoplamiento y cohesión.

- El acoplamiento entre dos módulos mide el nivel de asociación entre ellos; nos interesa buscar módulos poco acoplados
- La cohesión de un módulo mide el grado de conectividad entre los elementos que los forman; nos interesa buscar una cohesión alta

La **jerarquía** es un proceso de estructuración de varios elementos por niveles.

La programación orientada a objetos implementa estos cuatro conceptos con los siguientes elementos: clases y objetos, atributos y estado, métodos y mensajes, herencia y polimorfismo.

### Clases y Objetos

Una clase describe las estructuras de datos que lo forman y las funciones asociadas con él. Una clase es un modelo con el que se construyen los objetos.

Un objeto es un ejemplar concreto de una clase, que se estructura y comporta según se definió en la clase, pero su estado es particular e independiente del resto de ejemplares. Al proceso de crear un objeto se le llama generalmente **instanciar** una clase.

Las clases asumen el principio de encapsulación, se describe una vista pública que representa la funcionalidad de la misma, y una vista privada que describe los detalles de implementación.

Una clase es el único bloque de construcción, y por lo tanto, en una aplicación Java sólo hay clases; no existen datos sueltos ni procedimientos.

### **ATRIBUTOS Y ESTADO**

Un atributo es cada uno de los datos de una clase que la describen; no incluyen los datos auxiliares utilizados para una implementación concreta.

El estado de un objeto es el conjunto de valores de sus atributos en un instante dado.

### **MÉTODOS Y MENSAJES**

Un método define una operación sobre un objeto. En general, realizan dos posibles acciones: consultar el estado del objeto o modificarlo. Los métodos disponen de parámetros que permiten delimitar la acción del mismo.

Nos podemos encontrar con diversos tipos de métodos:

- Consultan o modifican un atributo, normalmente nos referenciaremos a ellos como: getters & setters
- Realizan operaciones sobre el conjunto de atributos, calculando valores o realizando modificaciones
- Inicializan los atributos al principio del ciclo de vida, o liberan los recursos al final del ciclo; nos referiremos a ellos como constructores o destructores

Un mensaje es la invocación de un método de un objeto. Podemos decir que un objeto lanza un mensaje (quien realiza la invocación) y otro lo recibe (el que ejecuta el método).

Podemos rescribir que una *clase* es la descripción e implementación de un conjunto de atributos y métodos.

### **HERENCIA Y POLIMORFISMO**

La herencia es una característica que permite a las clases definirse a partir de otras, y así reutilizar su funcionalidad. A la clase padre se le llama superclase, clase base..., y a la hija subclase, clase derivada....

El polimorfismo es la capacidad de que un mismo mensaje funcione con diferentes objetos. Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja. El método opera sobre un conjunto de posibles objetos compatibles.

## **Entorno de desarrollo Java**

El lenguaje Java fue creado por *Sun Microsystems Inc.*, Aparece en el año 1995 y debe, en gran medida, su popularidad al éxito del servicio WWW. Se creó en su origen para que fuese un lenguaje multiplataforma. Para ello se compila en un código intermedio: *bytecode* y necesita de una máquina virtual que lo ejecute. Normalmente, no utiliza código nativo, es decir, no se puede ejecutar directamente por el procesador.

Se disponen de varias plataformas Java para el desarrollo. Una plataforma es una combinación de hardware y software, usada para desarrollar y/o ejecutar programas.

Se va a empezar con el editor. Uno sencillo de libre distribución es el Notepad++:

<http://notepad-plus.sourceforge.net/es/site.htm>

Es un editor básico que reconoce la gramática del lenguaje; es recomendable para empezar con aplicaciones pequeñas, pero no para producir con efectividad.

Para el desarrollo y compilación de aplicaciones Java, utilizaremos: *Standard Edition* (Java SE) o *Java Development Kit* (JDK) de Sun:

Se trabaja mediante comandos en consola. Incluye, entre otras, las siguientes utilidades:

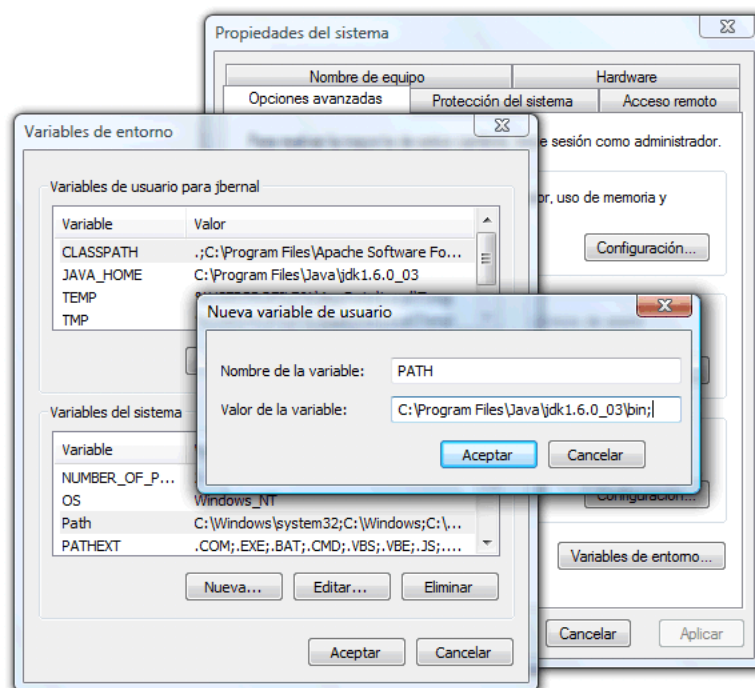
- El compilador: `javac.exe`. Un ejemplo: `javac Fich.java`
- El intérprete: `java.exe`. Ej. `java Fich`
- Un compresor: `jar.exe`. Ej. `jar -cvf fich.jar Uno.class Dos.class`
- El generador de documentación: `javadoc.exe`. Ej. `javadoc *.java`
- Un analizador de clases: `javap.exe`. Ej. `javap Fich`

Posteriormente, se verán plataformas más avanzadas como *Eclipse* o *Netbeans*.

Para la ejecución de aplicaciones Java (*Delivery Platforms*) es el *Java Runtime Environment* (JRE). Se instala automáticamente con Java SE.

Una vez instalado, debemos configurar la variable de entorno *PATH* para poder utilizar las utilidades desde la consola. Se debe añadir la ruta de la carpeta *bin*. En una instalación estándar podría ser:

```
C:\Program Files\Java\jdk1.6.0_16\bin;
```



**Figura 1. Java SE: variable de entorno PATH**

La variable de entorno *CLASSPATH* tiene especial importancia. Si no está definida busca las clases en la carpeta donde se ejecuta el *\*.class*. Si se define se debe incluir, además de los ficheros *\*.jar* de otras librerías, la ruta de la carpeta actual ("*./*"). Un ejemplo genérico sería:

```
CLASSPATH: .; c:\...\lib1.jar; ...\lib2.jar;
```

**Atención:** Para que tengan efecto los cambios en las variables de usuario hay que cerrar la consola o sesión.

## La consola

La consola es una ventana (llamada *Símbolo del Sistema* en Windows) que nos proporciona un punto de entrada para escribir comandos. Estos comandos nos permiten realizar tareas sin utilizar un entorno gráfico.

Aunque resulte un tanto incómodo, es una buena alternativa para la primera aproximación a Java.

Cuando se inicia la consola, aparece la ruta de una carpeta y acaba con el símbolo ">". Cualquier comando Java lo deberemos ejecutar sobre la carpeta de trabajo.

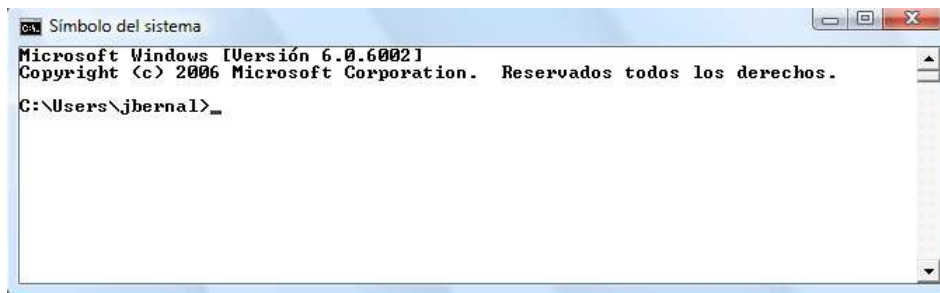


Figura 2. Java SE: consola

Sin profundizar en los diferentes comandos, veremos lo más básicos para poder realizar el trabajo con Java:

- Para cambiar de unidad, escribiremos la letra de unidad seguida de dos puntos, por ejemplo: “g:”
- cd. Con este comando cambiaremos de carpeta. Para subir un nivel escribiremos: “cd..”, y para profundizar “cd carpeta”
- dir. Lista el contenido de la carpeta
- type. Visualiza el contenido de un fichero de texto: “type fich.java”
- ↑, ↓. Repite comandos anteriores

Aunque existen muchos más comandos, el resto de acciones las realizaremos con entornos gráficos.

## El primer programa

Utilizaremos el *Notepad++* para crear un fichero de texto. Se debe llamar *HolaMundo.java*. Siempre que una clase sea pública, el fichero se debe llamar igual que la clase. Cuidado con las mayúsculas y minúsculas, son diferentes.

A continuación presentamos el contenido del fichero:

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("---- Hola Mundo ----");  
    }  
}
```

El siguiente paso es compilar el fichero *java* y obtener un fichero *class*. Para ello, y desde la consola, nos situaremos en el directorio que contiene el fichero *java*. Sería conveniente ejecutar el comando *dir* y observar que aparece nuestro fichero *java* *HolaMundo.java*. A continuación ejecutaremos en la consola:

```
javac HolaMundo.java
```

Si no se producen errores de compilación, se habrá creado un fichero *HolaMundo.class*. Ejecutar el comando *dir* para comprobarlo.

Por último debemos ejecutar el fichero *class*, para ello utilizaremos el comando siguiente:

```
java HolaMundo
```

Al ejecutarse se imprimirá por la consola el mensaje “--- Hola Mundo ---”.

Si entrar en muchos detalles, fijarse que hemos realizado una clase llamada *HolaMundo*. Esta clase tiene un sólo método (similar a procedimiento, función...) llamado *main*. Este método es importante porque es el único ejecutable desde la consola. La finalidad de nuestro método es imprimir un mensaje.

## Ficheros JAR

Los ficheros *Jar* (*Java ARchives*) permiten recopilar en un sólo fichero varios ficheros diferentes, almacenándolos en un formato comprimido para que ocupen menos espacio. Además, para utilizar todas

las clases que contiene el fichero *Jar* no hace falta descomprimirlo, sólo se debe referenciar mediante la variable de entorno *CLASPATH*.

El fichero *Jar* puede ser ejecutable, para ello se debe añadir un fichero manifiesto donde indicamos la clase principal; es un fichero de texto con extensión “*mf*” (un ejemplo sería *manifest.mf*) e introducimos una línea con el contenido: `Main-class: MiClasePrincipal`.

**Atención, se debe introducir un fin de línea (enter) para que funcione correctamente.**

```
Main-Class: MiClasePrincipal
```

Recordar situar la ruta de la consola en la carpeta de trabajo y realizar un `dir` para asegurarse de la existencia de los ficheros. Para crear el *Jar*, utilizamos el comando *jar*:

```
jar cvfm ejecutable.jar manifest.mf Uno.class Dos.class ...
```

A partir de aquí, podemos ejecutar el fichero *ejecutable.jar* como si fuera una aplicación normal, realizando un doble clic desde un entorno gráfico. Si la aplicación Java no trabaja con ventanas, que sería el caso de nuestro ejemplo *HolaMundo* anterior, se debe ejecutar desde consola:

```
java -jar ejecutable.jar
```

Si queremos añadir los ficheros fuente de la aplicación, normalmente se crea un fichero comprimido donde se incorpora la carpeta *src* con los fuentes y el fichero *jar*.

## Clase IO

La clase IO ha sido creada para facilitar la Entrada/Salida de datos en modo gráfico. Es una clase desarrollada propia y es independiente de la distribución Java SE. Se puede localizar en la ruta:

<http://www.eui.upm.es/~jbernal/io.jar>

En la actualidad, va por la versión 5.3.

Para su uso, se debe referenciar en la variable de entorno *classpath*. Además, se debe añadir la sentencia *import* en la clase:

```
import upm.jbb.IO;
public class HolaMundo {
    ...
}
```

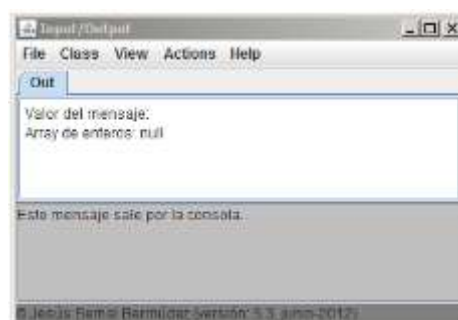


Figura 3. Clase IO

Los métodos más importantes son:

- Para leer un String: *IO.in.readString()* o *IO.in.readString("Mensaje")*
- Para leer un int: *IO.in.readInt()* o *IO.in.readInt("Mensaje")*
- Para leer un double: *IO.in.readDouble()* o *IO.in.readDouble("Mensaje")*
- Para leer cualquier clase: *IO.in.read("paquete.Clase", "Mensaje")*
- Para escribir un dato con salto de línea: *IO.out.println(...)*
- Para escribir un dato sin salto de línea: *IO.out.print(...)*
- Para escribir en la barra de estado: *IO.out.setStatusBar("mensaje")*

Para más información, en el menú *Help*, pulsar *Help Contents*

Un ejemplo de utilización de la clase IO es:

```
PruebaIO.java
import upm.jbb.IO;
public class PruebaIO {
    public static void main(String[] args) {
        String msg = IO.in.readString("Escribe un mensaje");
        IO.out.println("Valor del mensaje: " + msg);
        int entero = IO.in.readInt("Dame un número entero");
        double decimal = IO.in.readDouble("Dame un numero decimal");
        short corto = (short) IO.in.read("short", "Corto");
        Integer entero2 = (Integer) IO.in.read(8, "Valor por defecto");
        Byte b = (Byte) IO.in.read(new Byte("3"), "Valor por defecto");
        Integer[] intArray = (Integer[]) IO.in.read("Integer[]", "Array de enteros");
        IO.out.print("Array de enteros: ");
        IO.out.println(intArray);
        java.util.List<Byte> cb =
            (List<Byte>) IO.in.read("java.util.List<Byte>", "Colección");
        java.awt.Color color = (Color) IO.in.read("java.awt.Color", "color");
        IO.out.setStatusBar("Barra de estado");
        System.out.println("Este mensaje sale por la consola...");
    }
}
```

Podemos describir nuestro ejemplo de *HolaMundo* con la clase IO, el resultado es:

```
HolaMundo.java
import upm.jbb.IO;
public class HolaMundo {
    public static void main(String[] args) {
        IO.out.println("---- Hola Mundo ----");
    }
}
```

Para generar el fichero ejecutable, definiremos un fichero manifiesto. Se ha añadido una segunda línea para poder referenciar la librería *io.jar*:

```
manifest.mf
Main-Class: HolaMundo
Class-Path: ./io.jar
```

*Recordar insertar un salto de línea al final de HolaMundo.*

Y ejecutaríamos el comando jar:

```
jar cvfm hola.jar manifest.mf HolaMundo.class
```

Ahora, podremos realizar un doble clic a la aplicación *hola.jar* desde un entorno gráfico para ejecutarla, debe estar la librería *io.jar* en la misma carpeta.

## Lenguaje de programación Java

### Aspectos generales de la gramática

#### COMENTARIOS E IDENTIFICADORES

Los comentarios del código tienen tres formatos:

- Comentarios de línea: `“//”`, es comentario hasta final de línea
- Comentarios de bloque: `“/*” ... “*/”`

- Comentarios de documentación: “/\*\*” ...\*/”. Se utiliza para documentar los aspectos públicos de las clases; mediante la herramienta *javadoc.exe* se genera documentación en formato HTML con el contenido.

Los blancos, tabuladores y saltos de línea, no afectan al código.

Las mayúsculas y minúsculas son diferentes.

Los identificadores se forman mediante: {letra} + [letras | números | “\_” | “\$”].

Existe un estilo de nombrar los diferentes elementos ampliamente aceptado por la comunidad Java:

- **Clases:** las palabras empiezan por mayúsculas y el resto en minúsculas (ej. HolaMundo)
- **Métodos:** las palabras empiezan por mayúsculas y el resto en minúsculas, excepto la primera palabra que empieza por minúsculas (ej. holaMundo)
- **Constantes:** todo en mayúsculas separando las palabras con “\_”

## DECLARACIÓN DE VARIABLES Y CONSTANTES

Existen dos tipos de variables: las *primitivas* y las de *clase*. Las variables primitivas almacenan el dato en sí, por ello, dependiendo del tipo de dato ocupan un tamaño diferente de memoria. Las variables de clase almacenan la referencia del objeto y tienen un tamaño fijo de 32 bits.

La declaración de variables tiene el siguiente formato:

```
tipo identificador;
tipo identificador = expresión;
tipo identificador1, identificador2 = expresión;
```

Algunos ejemplos serían:

```
int prueba;
byte pruebaDos, prueba3 = 10, prueba4;
double prueba5 = 10.0;
```

Las constantes se declaran igual que las variables, pero añadiendo la palabra *final*.

```
final tipo IDENTIFICADOR = expresión;
```

Algunos ejemplos serían:

```
final int CONSTANTE_UNO = 10;
final double CONSTANTE_DOS = 10.0;
```

A continuación presentamos una tabla con los tipos primitivos, el tamaño que ocupa en memoria y el rango de valores que pueden representar.

- *Enteros*
  - *byte*. 8 bits. -128 a +127
  - *short*. 16 bits. -32.768 a +32.767
  - *int*. 32 bits. -2.147.483.648 a 2.147.483.648
  - *long*. 64 bits. -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
- *Decimales*
  - *float*. 32 bits.  $\approx \pm 1.7 \cdot 10^{38}$
  - *double*. 64 bits.  $\approx \pm 3.4 \cdot 10^{308}$
- *Caracteres*
  - *char*. 16. ‘\u0000’ a ‘\uffff’
- *Lógico*
  - *boolean*. true y false
- *Objetos*
  - *Object*

Vamos a realizar un breve adelanto de la clase String. Su utilidad es almacenar una cadena de caracteres. Un ejemplo de uso sería:

```
String nombre="Jorge", ciudad="Madrid";
```

Una cadena de caracteres no puede ser dividida mediante un salto de línea. Para cadenas largas, se dividen en varias y se concatenan con el operador “+”, pudiéndose definir en varias líneas.

## **PALABRAS RESERVADAS**

A continuación, se enumeran las palabras reservadas del lenguaje:

*abstract, assert, Boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, synchronized, switch, this, throw, throws, transient, try, void, volatile, while.*

## **LITERALES**

Los valores numéricos están en notación decimal. Para realizarlos en notación hexadecimal se añade “0x”, por ejemplo 0xffff. El tamaño por defecto de un entero es *int*, y de un decimal *double*. Para cambiar el tamaño a *long* se añade “L” al final (ej. 23L), para cambiar a *float* se añade “F”.

Los caracteres van entre comillas simples (ej. ‘x’). Los caracteres escape van precedidos por ‘\’, algunos ejemplos son:

- ‘\n’: retorno de carro
- ‘\t’: tabulador
- ‘\’: comillas simples
- ‘\”’: comillas dobles
- ‘\\’: propia barra
- ‘\b’: pitido
- ‘\u----’: valor Unicode

Los caracteres Unicode van precedidos por el carácter ‘u’ y en notación hexadecimal, algunos ejemplos serían: ‘\u0064’, ‘\u1234’.

## **OPERADORES**

Los operadores de Java son:

- *Asignación*
  - =. Ej.: x=y;
- *Aritméticos*
  - ++, --. Suma o resta 1, ej.: x++; ++x;
  - -. Cambio de signo, ej.: -x;
  - \*, /. Multiplicación y división, ej.: x\*y;
  - %. Resto de la división, ej.: x%y;
  - +, -. Suma y resta, ej.: x+2;
  - +=, -=, \*=, /=, %= . Ej.: x+=2; x=x+2;
- *Binarios*
  - ~. Not, ej.: ~x;
  - <<. Desplazamiento a la izquierda, se introducen ceros, ej.: x<<2;
  - >>. Desplazamiento a la derecha, se mantiene signo, ej.: x>>2;
  - >>>. Desplazamiento a la derecha introduciendo ceros
  - &, | y ^. And, or y xor, ej.: x&12;



- Comparación
  - ==, !=, <, >, <=, >=. Igual, distinto, menor que...
- Lógicos
  - !. Not lógico
  - &, &&, |, || y ^. And, and perezoso, or, or perezoso y xor

El operador “+” entre String realiza la función de concatenación.

## EXPRESIONES

Las expresiones son una combinación de operadores y operandos. La precedencia y asociatividad determina de forma clara la evaluación. El orden aplicado es:

- Paréntesis: ( )
- Unarios: + - !
- Multiplicativos: \* / %
- Aditivos: + -
- Relacionales: < > <= >=
- Igualdad: == !=
- Y lógico: & &&
- O lógico: | ||

## ARRAYS

Los arrays son tablas de valores del mismo tipo. En Java los arrays son dinámicos y deben crearse o reservar memoria en tiempo de ejecución para su utilización. Por lo tanto, para poder utilizar arrays primero se debe declarar, y posteriormente, reservar la memoria para su uso.

Para definir un array tiene el siguiente formato:

```
tipo[] identificador;
```

Se define pero no se crea, realmente el contenido de una variable de array es una referencia a una zona de memoria; el valor inicial es la palabra reservada *null*. Un ejemplo sería:

```
int[] diasMes;
```

Para reservar memoria para el array se debe indicar el tamaño; una vez establecido el tamaño, los posibles valores de índice van desde 0 a tamaño-1:

```
tipo[] identificador = new tipo[tamaño];
```

Si se vuelve a reservar memoria, se libera la memoria anteriormente utilizada y por lo tanto se pierde el contenido que tuviese.

Las dos acciones anteriores se pueden realizar en una sola línea:

```
int[] tabla = new int[10];
```

También se puede declarar un array, reservar memoria y darle contenido en una sola línea:

```
int[] tabla = { 0, 1, 2 };
```

Para saber el tamaño de un array no nulo (ya se ha reservado espacio en memoria) se puede utilizar el atributo *length*: `identificador.length`. Si se intenta consultar el tamaño de un array no creado da un error (*NullPointerException*).

Se pueden definir arrays de varias dimensiones. Por ejemplo:

```
int[][] matriz = new int[10][20];
```

Para acceder a una posición del array se realiza a través de su índice, tienen un rango fijo de 0 a tamaño-1. Algunos ejemplos son:

```
int i = tabla[2];  
i = matriz[2][3];
```

Si una variable de tipo array lleva el calificador *final*, quiere decir que el array no se permite que se vuelva a definir con el calificador *new*, pero si se permite cambiar el contenido del array. Mostramos un ejemplo para comprender este concepto:

```
final int[] array = new int[5];  
array[3] = 3; // Correcto  
// Error con array = new int[10];
```

## PROMOCIÓN Y CASTING

Cuando en Java operamos con expresiones, los resultados intermedios se almacenan en variables temporales con el tipo de mayor capacidad que intervenga en la operación; excepto con *byte* y *short* que se promociona automáticamente a *int*.

El *casting* es forzar la conversión de un tipo a otro. Esta conversión podría dar errores si son incompatibles. La forma de realizarlo es situando delante, y entre paréntesis, el nuevo tipo. Presentamos varios ejemplos:

```
int i = 3;  
byte b = (byte) i;  
byte b2 = 2;  
b2 = (byte) (b * 3);
```

Hay que tener cuidado que en las operaciones parciales no se pierda precisión. En la siguiente expresión: `int/int + double`, el valor parcial de la división se guarda en tipo *int*, y por lo tanto se pierden los decimales.

Para solucionar este problema tenemos dos opciones: `(double)int/int + double` o `1.0*int/int + double`.

Recordar que los valores enteros son, por defecto, *int*; y los valores decimales *double*.

## Sentencias

En general, todas las sentencias acaban con “;”.

### SENTENCIA DE BLOQUES

Agrupar en una sentencia de bloque un conjunto de sentencias. Para ello se sitúan entre llaves.

```
{ sentencia1; sentencia2; ...}
```

## Sentencias condicionales

### SENTENCIA IF

La sentencia *if* es una de las más básicas y nos permite controlar el flujo de control dependiendo de una condición. Los dos formatos posibles son:

```
if (condición) sentencia1;  
if (condición) sentencia1; else sentencia2;
```

Si en lugar de una sentencia tenemos varias, se debe utilizar la sentencia de bloque. Algunos ejemplos son:

```
if (a == 3) {  
    b = 8;  
}  
if (a > 2) {  
    a -= 1;  
    b += 1;
```

```

    }
    if (a > 2) {
        a -= 1;
        b += 1;
    } else a += 1;

```

Todas las sentencias se pueden anidar. Recordar que se debe aplicar un tabulador para su mejor comprensión. Un ejemplo sería:

```

    if (a > 2) {
        if (b == 8) {
            b++;
        }
    } else {
        b--;
    }

```

### SENTENCIA ELSE-IF

Aunque la sentencia *else-if* no existe como tal, queremos presentarla de forma independiente ya que debe tener un estilo peculiar de tabulación. Se considera una sentencia *else-if* si en la condición siempre se evalúa una misma variable o expresión; aunque existan anidamientos de sentencias, los tabuladores aplicados y el uso de llaves son diferentes a las normas generales. Un ejemplo sería:

```

    if (a==0) {
        sentencia1;
    } else if (a == 1 | a == 2) {
        sentencia2;
    } else if (a < 10) {
        sentencia3;
        sentencia4;
    } else {
        // No se debiera llegar aquí
    }

```

### SENTENCIA SWITCH

Esta sentencia evalúa una expresión, y dependiendo de su valor, ejecuta un conjunto de sentencias. Sólo se puede aplicar a valores primitivos, enumerados y a clases especiales. Su formato es:

```

    switch (Expresion) {
        case valor:
            sentencia1;
            sentencia2;
            break;
        case valor2:
        case valor3:
            sentencia3;
            break;
        case valor4:
            sentencia4; break;
        default: sentencia5;
    }

```

## Sentencias de bucles

### SENTENCIA WHILE Y DO WHILE

Estas sentencias se utilizan para realizar bucles o repeticiones hasta que se cumpla una determinada condición, pero a priori es indefinido el número de veces que se puede repetir.

La sentencia *while* tiene una repetición de 0 a n, y la sentencia *do while* de 1 a n. Su formato es:

```

    while (condición) {
        sentencia;
        sentencia;
        sentencia;
    }

```

```
do {
    senten2;
} while (condición)
```

## SENTENCIA FOR

Esta sentencia realiza un bucle, donde existe un índice de iteración y la condición de fin depende del índice. Tiene tres partes separadas por “;”: la primera se inicializan los índices (si hay más de uno se separan por “,”), la segunda es la condición de terminación (el bucle se repite mientras se cumpla la condición) y la tercera es el incremento de los índices.

Si los índices se declaran en el propio *for*, no se podrán utilizar fuera de éste. Normalmente, la condición debe depender del valor del índice.

Su formato es:

```
for (inicialización; terminación(mientras); operación) {
    sentencial;
    ...
}
```

Mostramos algunos ejemplos:

```
for (int i = 1; i <= 10; i++) {
    a *= 2;
}

int i, j;
for (i = 1, j = 1; (i <= 5) && (j <= 5); i++, j += 2) {
    // ...
}

for (;;) {
    // ...
} // bucle infinito
```

## SENTENCIA CONTINUE Y BREAK

Con la sentencia *continue*, se continúa con la siguiente interacción del bucle; puede aparecer en varios lugares del bucle, pero normalmente asociado a una condición.

Con la sentencia *break* abortamos el bucle, o en general, el bloque donde se ejecute.

Es posible que necesitemos un bucle con índice y además una condición especial de finalización, en este caso, se recomienda el bucle *for* con la sentencia *break* para la condición especial.

Un ejemplo sería:

```
int a = 1;
for (int i = 1; i <= 100; i++) {
    if (a % 5 == 0) continue;
    if (a > 1000) break;
    a += a * 2 + 1;
}
```

## SENTENCIA FOR EACH

Esta sentencia es útil para procesar una colección de datos, donde sólo nos interesa el conjunto de datos y no el orden con que se procesan.

Se puede aplicar a las clases *Collection* y a los *arrays*. Su formato es:

```
for (int item: array){
    // en la variable item tenemos el dato
}
```

Un posible ejemplo sería:

```
int[] datos = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (int item : datos) {
    System.out.println(item);
}
```

Es equivalente al siguiente bucle:

```
for (int i=0; i<datos.length;i++) {
    System.out.println(datos[i]);
}
```

✍ Realizar la clase *BucleBasicoTriple*. A partir de un valor leído mediante IO (le llamaremos *n*), escribir la secuencia 0, 1, 2... hasta *n* tres veces, la primera realizada con la estructura *while*, la segunda con *do...while* y la tercera con *for*.

✍ Realizar la clase *Serie1*. Dado un valor leído mediante IO (*n*), calcular el sumatorio:  $n + (n-1) + (n-2) + \dots + 1$ ; por ejemplo, para  $n=4$  sería  $4+3+2+1 = 10$ .

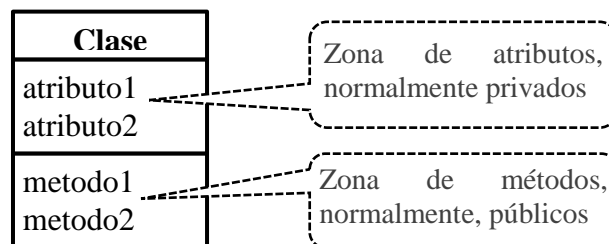
✍ Realizar la clase *Tabla*. Se define una array con el contenido {3,2,6,8,4}, calcular la suma de todos los elementos dos veces, una mediante la estructura *for* y otra mediante *for each*.

✍ Realizar la clase *Tabla2*. Se define una array con el contenido {3,2,6,8,4}, ordenar el array de menor a mayor.

## Clases y objetos

### Clases: atributos y métodos

Como ya definimos anteriormente, una clase es un conjunto de atributos y métodos. Vemos en la figura siguiente, una manera de mostrar una clase.



Las clases tienen dos vistas: una pública y una privada. La vista pública, también llamada interfaz, establece los aspectos visibles de la clase, y por lo tanto, nos ofrece los métodos que podemos utilizar. La vista privada, también llamada implementación, encapsula todos los detalles de la estructura de datos y la implementación de los métodos.

### Atributos

Normalmente, los atributos son privados, y por lo tanto, no son directamente accesibles. La consulta o modificación de los atributos se debe realizar a través de métodos públicos; de esta manera, el objeto controla las operaciones válidas que se pueden realizar.

Un atributo es una información que se almacena y es parte representativa del estado. Los atributos se pueden sustentar mediante tipos primitivos o clases. Mostremos algunos ejemplos.

La clase *Punto* representa un punto en un espacio de dos dimensiones, sus atributos son *x* e *y*. La declaración de la clase quedaría:

```
public class Punto {  
    private int x, y;  
}
```

Observar, que los atributos se declaran a nivel de instancia, fuera de los métodos, y por lo tanto, son accesibles de cualquier parte de la clase.

La clase *NumeroNatural* representa un valor natural, su atributo podría ser *valor*. La clase quedaría:

```
public class NumeroNatural{  
    private int valor;  
}
```

La clase *Usuario* representa a un usuario, podría tener los atributos *nombre*, *eMail*, *movil*, *ciudad* y *mayorEdad*. El código java sería:

```
public class Usuario{  
    private String nombre, eMail, ciudad;  
    private long movil;  
    private boolean mayorEdad;  
}
```

Como puede observarse en todos los ejemplos, los atributos se declaran a nivel de la clase y están calificados con *private*, indicando que no es accesible desde fuera de la clase.

Se pueden crear tantos objetos o instancia de las clases como queramos. Dos objetos de la clase *Punto* tienen valores de atributos independientes y evolucionan por separado; eso sí, cuando se crea la instancia, todas parten del mismo estado.

## Métodos

La razón de establecer los atributos en vista privada, es poder controlar la modificación del estado del objeto y no permitir que este evolucione hacia estados incoherentes. Pero debemos establecer los mecanismos para poder modificar el objeto, y son los métodos, el sistema adecuado.

Podemos organizar los métodos en tres tipos, teniendo en cuenta aspectos sintácticos y semánticos:

- Constructores. Métodos que inicializan la instancia
- Métodos genéricos. Realizan acciones utilizando los atributos
- Métodos para accesos directo a los atributos (*getters & setters*). Estos métodos son opcionales y suelen utilizarse más en clases de tipo “entidad”, es decir, que tienen persistencia en bases de datos, o en los “beans”, donde tienen una correspondencia con formularios. Aunque estos métodos son realmente generales, los distinguimos por separado por tener unas normas concretas de estilo.

### MÉTODOS GENÉRICOS

Comencemos con el formato básico de un método:

```
[public | private] {void|tipo} identificadorMetodo (tipo param1, tipo param2...) {...}
```

El primer calificador es *public* o *private*, se establece el tipo de vista.

La segunda palabra es el tipo devuelto, con el calificador *void* (vacío) se indica que no devuelve nada, o se establece el tipo devuelto.

La siguiente palabra representa el nombre del método.

A continuación vienen los parámetros. Hay que resaltar, en general, los datos con que actúan los métodos son sus propios atributos, y por lo tanto no es necesario pasarlos por parámetro. Por parámetros se pasan valores externos al objeto que condicionan la acción a realizar.

Existen dos tipos de parámetros:

- Tipos primitivos (byte, short, int...). Estos se pasan por *valor*, es decir, se realiza una copia del contenido en la variable que sustenta el parámetro, y por lo tanto, si el método modifica el parámetro no afecta a la variable del mensaje entrante.
- Clases y arrays. Estos se pasan por *referencia*, es decir, se pasa la dirección en el parámetro, así que, modificaciones dentro del método afectan externamente.

Con la sentencia *return*, se finaliza el método. Si además queremos devolver un valor se utiliza la sentencia *return valor*.

Ampliando la clase Punto, se podrían añadir los siguientes métodos:

```
public class Punto{
    private int x, y;
    public double modulo() {
        double valor = 0;
        // Se calcula el modulo del vector con x,y
        return valor;
    }
    public double fase() {
        double valor = 0;
        // Se calcula la fase del vector con x,y
        return valor;
    }
}
```

Dentro de los métodos podemos definir variables locales, las cuales serán creadas en la ejecución del método y destruidas al finalizar la ejecución. Las variables de método, sólo son visibles desde el propio método en que se definen.

### MÉTODOS PARA ACCESOS A ATRIBUTOS (GETTERS & SETTERS)

Realmente, a nivel gramatical, se pueden considerar como métodos genéricos. Pero semánticamente, tienen connotaciones especiales que debemos especificar.

Cada atributo tiene dos posibles acciones: leer su valor o establecerlo. No tenemos por qué realizar ambas acciones, depende del diseño que se busque, de hecho, muchas clases no tienen estos métodos.

Existe una nomenclatura especial para el identificador del método:

- Establecer el valor de un atributo: *set + NombreAtributo* (la primera letra del nombre del atributo debe estar en mayúsculas). Por ejemplo, en la clase *Punto* sería *setX* y *setY*, en la clase *NumeroNatural* sería *setValor*, en la clase *Usuario* sería *setNombre*, *setEmail*, *setCiudad*, *setMovil* y *setMayorEdad*.
- Leer el valor del atributo: *get + NombreAtributo* o *is + NombreAtributo* si devuelve un tipo *boolean*. Por ejemplo, en la clase *Punto* sería *getX* y *getY*, en la clase *NumeroNatural* sería *getValor*, en la clase *Usuario* sería *getNombre*, *getEmail*, *getCiudad*, *getMovil* y *isMayorEdad*.

A continuación, presentamos la estructura completa del código de la clase Punto:

```
Punto.java
public class Punto{
    private int x,y;
    public void setX(int x){
        //...
    }
    public void setY(int y){
        //...
    }
    public int getX(){
        return x;
    }
    public int getY(){
        return y;
    }
    public double modulo(){
        //...
    }
    public double fase(){
        //...
    }
}
```

```
}  
}
```

## CONSTRUCTORES

Los constructores son métodos especiales que reúnen las tareas de inicialización de los objetos de una clase; por lo tanto, el constructor establece el estado inicial de todos los objetos que se instancian.

No es obligatorio definir constructores, si no se realiza, existe un constructor por defecto sin parámetros.

La ejecución del constructor es implícita a la creación de una instancia.

La restricción sintáctica del constructor es que debe llamarse igual que la clase y no devuelve ningún tipo ni lleva la palabra *void*.

Como ejemplo añadimos un constructor a la clase Punto:

```
public class Punto{  
    private int x,y;  
    public Punto(int x, int y){}  
    ...  
}
```

## SOBRECARGA DE MÉTODOS

La sobrecarga es definir dos o más métodos con el mismo nombre, pero con parámetros diferentes por cantidad o tipo. El objetivo de la sobrecarga es reducir el número de identificadores distintos para una misma acción pero con matices que la diferencian. La sobrecarga se puede realizar tanto en métodos generales, como en constructores.

La sobrecarga es un polimorfismo estático, ya que es el compilador quien resuelve el conflicto del método a referenciar.

Si definimos un constructor con parámetros, el constructor sin parámetros deja de estar disponible; así que, si nos interesa, se debe definir para su utilización.

Como ejemplo añadimos sobrecarga a los constructores de la clase Punto:

```
public class Punto{  
    private int x,y;  
    public Punto(int x, int y){}  
    public Punto(int xy){}  
    public Punto(){}  
    ...  
}
```

## Objetos

Recordemos que la clase representa el modelo para crear objetos, pero son los objetos los elementos con los que podemos trabajar. Los objetos ocupan memoria y tiene un estado, que representa el conjunto de valores de sus atributos.

Podemos crear todos los objetos que queramos, todos ellos tiene existencia propia, pudiendo evolucionar por caminos diferentes e independientes.

Cuando se crea un objeto se debe asociar con una referencia, por lo tanto, el proceso debe tener dos partes:

1. Declarar una variable que pueda referenciar al objeto en cuestión
2. Crear el objeto y asociarlo a la variable. Para crear el objeto, se realiza con la sentencia **new**, recordar que lleva implícito llamar al método constructor para que inicialice el objeto

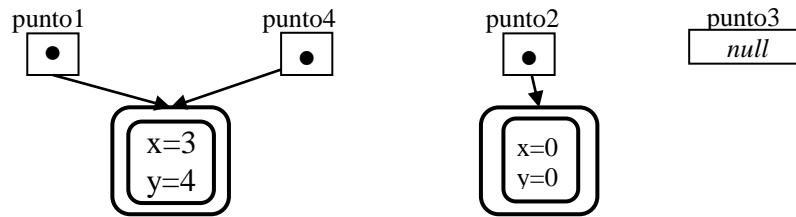
Un ejemplo sería:

```
Punto punto1, punto2 = new Punto(), punto3, punto4;  
punto1 = new Punto(3,4);
```



```
punto4 = punto1;
```

En este ejemplo declaramos cuatro puntos de la clase Punto. Cada vez que se utiliza la palabra **new**, quiere decir que se crea una nueva instancia, por lo tanto, tenemos un total de dos instancias. En el ejemplo se han utilizado dos constructores diferentes para inicializar las instancias. Mostramos a continuación, una gráfica una vez ejecutado todo el código.



Fijarse que las variables *punto1* y *punto4* referencian la misma instancia, es una manera redundante de acceder. El contenido de una variable de instancia es la dirección de memoria donde se encuentra el objeto. Cuando comparamos variables de instancia, realmente comparamos direcciones de memoria.

Se ha declarado la variable *punto3*, pero no se ha asociado a ninguna instancia, en ese caso, su contenido es *null*. Podemos asignar a cualquier variable de instancia el valor *null*, por ejemplo *punto2=null*.

En Java, no existe ninguna forma explícita para destruir un objeto, simplemente cuando un objeto se queda huérfano (no tiene ninguna referencia) se destruye. Esta labor la realiza el recolector de basura (*garbage collector*) de forma automática. En el ejemplo anterior, si asignamos a *punto2* un *null*, queda su instancia sin referencia, por consiguiente, cuando se lance el recolector de basura destruirá el objeto huérfano.

Notar que cuando creamos un array NO implica la creación de los objetos. Veamos un ejemplo en tres pasos para entender mejor el proceso.

- En primer lugar se declara una variable de tipo array de Punto.

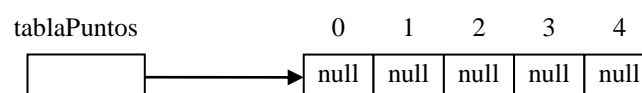
```
Punto[] tablaPuntos;
```

El contenido de la variable *tablaPuntos* es *null*.

- En segundo lugar se crea el array.

```
tablaPuntos = new Punto[5];
```

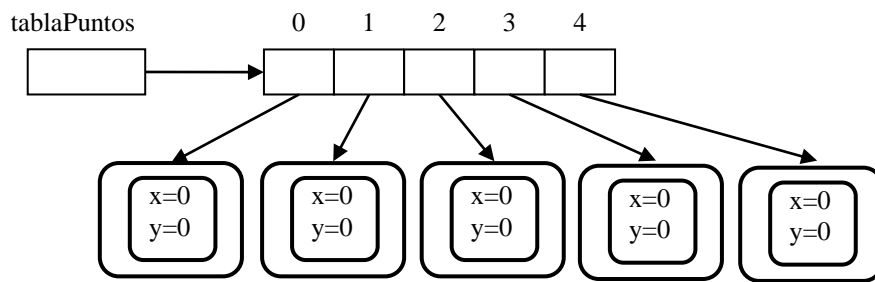
Gráficamente se describiría como:



- En tercer lugar se crean los objetos.

```
for (int i = 0; i < 5; i++)
    tablaPuntos[i] = new Punto();
tablaPuntos[i] = new Punto();
```

Gráficamente se describiría como:



Si una variable de Clase lleva el calificador *final*, quiere decir que no se permite que se vuelva a crear una nueva instancia con el calificador *new*, pero si se permite cambiar el contenido de la instancia. Mostramos un ejemplo para comprender este concepto:

```
final Punto pto = new Punto();
pto.setPtX(3); //Correcto
//Error con pto = new Punto();
```

## Mensajes

Una vez creado los objetos es posible pasarles mensajes, o invocarles métodos, para solicitar una acción. El objeto establece a que método le corresponde el mensaje recibido.

Para enviar un mensaje se realiza con la notación *punto*. Dependiendo si almacenamos el valor devuelto del método, tenemos dos posibilidades:

```
referencia.metodo(param1,param2...);
referencia.metodo();

var = referencia.metodo(param1,param2...);
var = referencia.metodo();
```

La segunda línea es para guardar el valor devuelto por el mensaje.

Ponemos a continuación un ejemplo con la clase Punto:

```
Punto punto1 = new Punto(2,5);
punto1.setX(4);
int x = punto1.getX();
double d = punto1.modulo();
punto1.setX(punto1.getX() + 1); // suma en una unidad el valor de x
```

## THIS

Dentro de la implementación de un método, a veces se necesita referenciar a la propia instancia a la cual pertenece. Para ello está la palabra reservada *this*. Un ejemplo dentro de *Punto* sería: *this.x = 3*.

Resulta recomendable calificar con *this* a todas las referencias a los atributos y métodos de la propia clase; y resulta obligatorio cuando queramos distinguir entre un parámetro y una variable de instancia con el mismo nombre.

También referenciamos con la palabra *this* a nuestros propios constructores, por ejemplo *this(3,4)*.

A continuación presentamos el código completo de la clase Punto:

```
Punto.java
public class Punto {
    private int x, y;
    public Punto(int x, int y) {
        this.setX(x);
        this.setY(y);
    }
    public Punto(int xy) {
        this(xy, xy);
    }
    public Punto() {
        this(0, 0);
    }
}
```

```

    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public double modulo() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
    public double fase() {
        double aux = (double) (this.y) / this.x;
        return Math.atan(aux);
    }
}

```

Se debe evitar la redundancia de código, para ello sólo se implementa un solo constructor y el resto referencian a este, mediante valores por defecto de los parámetros. Los constructores deben hacer uso de los métodos para el acceso a los atributos, y así, se aplica los filtros o controles pertinentes.

Otro ejemplo es la clase *NumeroNatural*:

```

NumeroNatural.java
public class NumeroNatural {
    private int natural;
    public NumeroNatural(int natural) {
        this.setNatural(natural);
    }
    public NumeroNatural() {
        this(0);
    }
    public int getNatural() {
        return this.natural;
    }
    public void setNatural(int natural) {
        if (natural < 0) this.natural = 0;
        else this.natural = natural;
    }
    public void añadir(int valor) {
        //this.natural += valor; //Error se salta el filtro
        this.setNatural(this.natural + valor);
    }
}

```

 **Realizar la clase *Rango* y *PruebaRango*, representa un número natural limitado hasta un máximo.**

- Atributos:
  - valor (lectura y escritura). En la escritura se deben filtrar los valores fuera de rango
  - maximo (sólo lectura). Sólo lectura significa que el método *setMaximo* está calificado con *private* o, simplemente, no se define el método.
- Constructores:
  - *Rango(int maximo, int valor)*
  - *Rango()*
- Métodos:
  - void *reset()*, pone *valor* a cero

✍ Realizar la clase *ContadorCircular* y *PruebaContadorCircular*, representa un contador similar a un cuentakilómetros de un coche, con cada petición incrementa en uno el contador y al llegar al valor máximo vuelve a cero.

- Atributos:
  - maximo (sólo lectura) representa el valor máximo de cuenta, sólo se puede establecer con el constructor
  - contador (sólo lectura)
- Constructores:
  - ContadorCircular(int maximo)
- Métodos:
  - void reset(), pone a cero el contador
  - void incrementar(), incrementa en una unidad el contador. Cuando se pase del máximo pasa automáticamente a cero

## Elementos estáticos

Todos los conceptos expuestos giran en torno a una idea principal: la clase describe el comportamiento de cada uno de sus objetos.

El presente apartado pretende dar solución a cuestiones de la clase y de todos los objetos en su generalidad. Por ejemplo, si definimos una constante de una clase que tenga siempre el mismo valor, es común a todos los objetos, y por lo tanto, debe estar asociado a la clase y no a las instancias.

Podemos definir atributos estáticos y métodos estáticos.

### ATRIBUTOS ESTÁTICOS

Para definirlos se antepone al tipo la palabra *static*; se deben inicializar independientemente a los constructores.

Estos atributos están compartidos por todos los objetos de la clase y son accesibles desde cada uno de ellos. Cualquier modificación por parte de un objeto afectará al resto.

Para el acceso a un atributo, en lugar de utilizar *this*, se utiliza el nombre de la clase: *Clase.atributo*.

Normalmente, las constantes asociadas a la clase son buenas candidatas para realizarlas estáticas.

Un ejemplo sería:

```
public class MiClase{
    private static final int MAX = 100;
    //...
}
```

### MÉTODOS ESTÁTICOS

Al igual que los atributos, se antepone la palabra *static* al tipo devuelto.

No puede utilizar en el código la palabra *this*, ya que está asociado exclusivamente a las instancias. Se pueden acceder a los atributos estáticos del mismo modo que pueden las instancias, anteponiendo el nombre de la clase: *Clase.atributo*.

Se permite definir un código estático para inicializar los atributos estáticos. Su ejecución se dispara una sola vez al principio. Tiene el siguiente formato:

```
static {
    //...
}
```

Vamos añadir algunos aspectos estáticos a la clase Punto:

#### Punto.java

```
public class Punto {
    public static final double PI = 3.1415;
    private static int numeroInstancias;
    private int x, y;
    static {
        Punto.numeroInstancias = 0;
    }
    //Constructores. Deben incrementar el atributo numeroInstancias
    public static int instanciasCreadas() {
        return Punto.numeroInstancias;
    }
    //...
}
```

#### PruebaPunto.java

```
public class PruebaPunto {
    public static void main(String[] args){
        Punto p;
        p = new Punto(2,2);
        p = new Punto(2,2);
        p = new Punto(2,2);
        System.out.println(Punto.instanciasCreadas());
        System.out.println(Punto.PI);
    }
}
```

## Clases de Java fundamentales

### CLASE STRING

La clase String representa una cadena de caracteres. Su contenido no puede cambiar, por lo tanto, cada vez que se modifica, se crea un nuevo String. Consume menos recursos para el almacenamiento pero las operaciones de modificación resultan costosas en el tiempo. Si van a existir numerosas modificaciones se deberá utilizar la clase StringBuffer.

Todos los literales de cadena como “abc” se implementan como una instancia de String, podemos realizar la siguiente asignación:

```
String s = "abc";
```

Si se quiere crear cadena a partir de valores o variable nativas, debemos utilizar el método estático valueOf(...). Algunos ejemplos son:

```
String s;
s = String.valueOf(3);
s = String.valueOf(3.3);
s = String.valueOf('a');
s = String.valueOf(true);
```

El operador “+” entre cadenas se interpreta como concatenación. De forma automática, al concatenar un String con cualquier otro valor, lo convierte a String. Algunos ejemplos son:

```
String s = "abc", s2;
int i = 1;
s2 = s + i; // "abc1"
s2 = "" + 3.4; // "3.4"
```

La clase String tiene un gran número de métodos que nos facilita el trabajo con cadenas. Algunos de los más útiles son:

- Longitud de la cadena (*length*).
- Comparar cadenas (*equals*, *equalsIgnoreCase*, *compareTo*).
- Busca subcadenas (*indexOf*, *lastIndexOf*)
- Extrae subcadenas (*substring*, *toLowerCase*, *toUpperCase*, *trim*)

Algunos ejemplos de uso son:

```
String s = "abcde";
int l = s.length(); // 5
String s2 = s.toUpperCase(); // ABCDE
int i = s.indexOf("cd"); // 2
String s3 = s.substring(1, 3); // bc
```

## CLASE *StringBuffer*

Representa una cadena de caracteres con una estructura interna que facilita su modificación, ocupa más que la clase *String*, pero las operaciones de modificación son muchos más rápidas.

Para la modificación del contenido, cuenta con los métodos *insert* y *append*. Mirar el API para mayor documentación.

En el siguiente código, el bucle realizado con *StringBuffer* es aproximadamente 1000 veces más rápido que el bucle realizado con *String*. Es importante tener en cuenta, que cuando se requiera modificar una cadena de forma dinámica, es mucho mejor la clase *StringBuffer*.

```
String s = "";
StringBuffer sb = new StringBuffer();

for (int i = 0; i < 100000; i++) {
    s += ".";
}

for (int i = 0; i < 100000; i++) {
    sb.append(".");
}
```

## CLASE *NUMBER*: *Integer*, *Double*...

Este conjunto de clases son útiles para trabajar con los valores primitivos, pero mediante objetos. A estas clases se les llama *Wrappers* (envolventes). Cada clase especializada, almacena un tipo de valor primitivo numérico.

Las clases tienen constantes que nos pueden ser útiles, por ejemplo, todas tienen *MAX\_VALUE* y *MIN\_VALUE*, *Double* además tiene *NaN*, *POSITIVE\_INFINITY*, *NEGATIVE\_INFINITY*...

Otra utilidad interesante es que pueden convertir una cadena a un valor numérico, por ejemplo, *Integer.parseInt("123")* devuelve el valor *int* asociado.

Para facilitar su uso, en las versiones actuales de Java existe una conversión automática entre el tipo primitivo y el objeto (*boxing* y *unboxing*). El siguiente código funciona correctamente:

```
Integer claseI;
Double claseD;
int i;
double d;
claseI = 23;
claseD = 2.2;
i = claseI;
d = claseD;
```

## *BOOLEAN*, *CHARACTER*, *MATH*, *SYSTEM*...

Mirar el API para mayor documentación.

## Javadoc

Los comentarios *JavaDoc* están destinados a describir, principalmente, las clases y los métodos. Existe una utilidad de Sun Microsystems llamada *javadoc.exe* para generar APIs en formato HTML de un documento de código fuente Java.

Recordar que los comentarios *javadoc* son `/** ... */`. Se utiliza el carácter `@` como anotación especial, para describir las etiquetas de documentación.

Los comentarios se deben colocar justo antes del elemento a documentar.

Se permite añadir elementos HTML a la documentación para enriquecer su presentación.

## ANOTACIONES

### Generales

- **@see referencia.** Realiza un enlace. Ejemplos de referencia son: `#metodo()`; `clase#metodo()`; `paquete.clase`; `paquete.clase#metodo()`; `<a href="..."> enlace estándar de html </a>`
- **@since.** Indica en la versión que apareció este elemento respecto a la secuencia de versiones aparecidas a lo largo del tiempo

### Paquetes

- **@see, @since**
- **@deprecated.** Indica que este elemento es antiguo y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores. Se debe indicar la alternativa correcta

### Clases e interfaces

- **@see, @since, @deprecated**
- **@version.** Versión y fecha
- **@author.** Nombre del autor

### Variables y constantes

- **@see, @deprecated**

### Métodos

- **@see, @since, @deprecated**
- **@param.** Definición de un parámetro, es obligatorio. `*@param parámetro descripción`
- **@return.** documenta el dato devuelto, es obligatorio. `*@return descripción`
- **@throws.** Excepción lanzada por el método. `*@throws clase descripción`

Un ejemplo de uso podría ser:

```
Mes.java
/**
 * La clase Mes representa un mes. Esta clase puede trabajar con valores
 * enteros: 1..12, o con strings: "enero".."diciembre".
 *
 * @author Jesús Bernal Bermúdez
 * @version 1.0 04/03/2008
 * @since 1.0
 */
public class Mes {
    /**
     * Array con los strings de los doce meses
     */
    public static final String[] MESES = { "enero", "febrero", "marzo", "abril", "mayo",
        "junio", "julio", "agosto", "septiembre", "octubre", "noviembre", "diciembre" };
    private int mes; // Tipo primitivo que almacena el mes de la instancia.

    /**
     * Crea una nueva instancia a partir de un valor entero.
     *
     * @param mes
     *      int que representa el mes.
     */
    public Mes(int mes) { ... }

    /**
     * Crea una nueva instancia y se inicializa con el mes 1, "enero".
     */
    public Mes() { ... }
```

```

/**
 * Crea una nueva instancia a partir de un string. Si no se reconoce el
 * string se interpreta "enero"
 *
 * @param mes
 *         String que representa el mes. No se distinguen mayúsculas de
 *         minúsculas.
 */
public Mes(String mes) {...}

/**
 * Establece un nuevo mes.
 *
 * @param mes
 *         int que representa el mes.
 */
public void setMes(int mes) {...}

/**
 * Establece un nuevo mes a partir de un string. Si no se reconoce el string
 * se interpreta "enero"
 *
 * @param mes
 *         String que representa el mes. No se distinguen mayúsculas de
 *         minúsculas.
 */
public void setMes(String mes) {...}

/**
 * Devuelve el mes que representa la instancia en formato numérico.
 *
 * @return int que representa el mes.
 * @see #getMes()
 */
public int getMes() {...}

/**
 * Devuelve el mes que representa la instancia en formato string.
 *
 * @return String que representa el mes.
 */
public String getMesString() {...}

/**
 * Se pasa al mes siguiente.
 */
public void incrementa() {...}

/**
 * Devuelve un String que reprersenta el objeto.
 *
 * @return un String que representa al objeto
 */
public String toString() {...}

/**
 * Devuelve una nueva instancia que es una copia de la actual.
 *
 * @return Mes que representa un clon de la actual
 */
public Mes clonar() {...}

/**
 * Devuelve el string correspondiente a un entero. Si el valor entero no es
 * correcto se devuelve null.
 *
 * @param valor
 *         int que representa el mes a convertir.
 * @return String asociado al mes.
 */
public static String literalDe(int valor) {...}

/**
 * Devuelve el entero correspondiente a un mes.No se distinguen mayúsculas de
 * minúsculas. Si no se reconoce el String se devuelve 0.
 *
 * @param valor
 *         String que representa el mes a convertir.

```



```

    * @return int asociado al mes.
    */
    public static int valorDe(String valor) {...}
}


```

Para generar la documentación en *html*, se realiza de forma parecida a compilar, desde la consola se teclea: `javadoc Mes.java`. Conviene realizarlo en una carpeta independiente debido a la gran cantidad de ficheros que genera. Si queremos realizarlos con varias clases sería:

```
javadoc C1.java C2.java C3.java
```

```
javadoc *.java
```

### *Implementa por completo la clase Mes*

 *Realizar la clase Pila, gestiona una pila de cadenas (String) (último en entrar, primero en salir). Para su implementación se apoya en un array.*

- Atributos:
  - capacidad (sólo lectura) representa el tamaño del array. Sólo se puede establecer en el constructor
  - array (sin acceso). Significa que los métodos get/set son *private* o no se implementan
  - cima (sin acceso) apunta al último elemento apilado, -1 si no hay elementos
- Constructores:
  - Pila(int capacidad), Pila()
- Métodos:
  - void borrarTodo(), borra toda la pila
  - boolean apilar(String elemento). Añade un elemento en la cima de la pila, si no está llena. Devuelve un valor lógico indicando si se ha podido añadir
  - String desapilar(). Devuelve el String que se encuentra en la cima de la pila, devuelve null si la pila está vacía
  - int tamaño(). Devuelve el número de String que contiene la pila

La clase *TestPila* debe tener el siguiente contenido:

```

TestPila.java
import upm.jbb.IO;

public class PilaTest {
    private Pila pila;

    public PilaTest() {
        pila = new Pila(3);
        IO.in.addController(this);
    }

    public void borrarTodo() {
        pila.borrarTodo();
        IO.out.println(">>>" + pila.toString());
    }

    public void apilar() {
        IO.out.println(">>> " + pila.apilar(IO.in.readString("Apilar")) +
            ">>>" + pila.toString());
    }

    public void desapilar() {
        IO.out.println(">>> " + pila.desapilar() + ">>>" + pila.toString());
    }

    public void tamaño() {
        IO.out.println(">>> " + pila.tamaño());
    }
}

```

```

    }

    public static void main(String[] args) {
        new PilaTest();
    }
}

```

La clase IO, mediante el método *addController()*, crea un botón por cada método que aparezca en la prueba. Estos métodos devuelven *void* y no pueden tener parámetros. Así, podremos probar de una manera gráfica todos los métodos de nuestra clase.

## Herencia y polimorfismo

### Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos, por medio del cual una clase se construye a partir de otra. Una de sus funciones más importantes es proveer el polimorfismo.

La herencia relaciona las clases de manera jerárquica; una clase padre, superclase o clase base sobre otras clases hijas, subclases o clase derivada. Los descendientes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables, además de crear los suyos propios.

En Java, sólo se permite la herencia simple, o dicho de otra manera, la jerarquía de clases tiene estructura de árbol. El punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases. Para especificar la superclase se realiza con la palabra *extends*; si no se especifica se hereda de *Object*. Algunos ejemplos son:

```

public class Punto {...} //se hereda de Object
public class Punto extends Object {...} //es lo mismo que la anterior
public class Punto3D extends Punto {...}

```

Cuando decimos que se heredan los miembros (atributos y métodos) de la superclase, quiere decir que se pueden referenciar con respecto a la clase hija. Por ejemplo, si la clase *SubClase* hereda de *SuperClase*, y esta tiene un método llamado *mtd1*, el siguiente mensaje es correcto:

```
subClase.mtd1();
```

En este caso, la clase hija recibe el mensaje, y esta dispone de la funcionalidad por la herencia de la clase padre. Para que un atributo o método, sea directamente utilizable por la clase hija, debe ser *public*, *protected* o *friendly* (hasta ahora sólo hemos hablado de *public*, en el tema de paquetes aclararemos estos calificadores). Los miembros que sean *private*, no se podrán invocar directamente mediante *this*, pero su funcionalidad esta latente en la herencia.

Para referirnos a los miembros de la clase padre lo realizaremos mediante la palabra *super*. Con “*super.*” accedemos a los miembros y con “*super()*” accedemos a los constructores.

Los constructores no se heredan, aunque existe llamadas implícitas a los constructores de las clases padre. Al crear una instancia de una clase hija, lo primero que se hace es llamar al constructor de la clase padre para que se inicialicen los atributos de la clase padre, sino se especifica, se llama al constructor sin parámetros. Por ello, si se quiere invocar otro constructor, la primera sentencia del constructor de la clase hija debe ser la llamada al constructor de la clase padre.

Veamos un ejemplo completo. Vamos a partir de la superclase *Punto*, que si no se especifica su superclase, hereda de *Object*.

```

public class Punto {
    private int x, y;

    public Punto(int x, int y) {
        this.setX(x);
        this.setY(y);
    }
}

```

```

public Punto() {this(0, 0);}

public final void setX(int x) {this.x = x;}
public final void setY(int y) {this.y = y;}
public int getX() {return this.x;}
public int getY() {return this.y;}

public double modulo() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
}
}

```

Ahora queremos crear una clase *Punto* pero que tenga en cuenta el tiempo, le llamaremos *PuntoTiempo*.

```

PuntoTiempo.java
public class PuntoTiempo extends Punto {
    private int t;

    public PuntoTiempo() {
        this(0,0,0) ;
    }
    public PuntoTiempo(int x, int y, int t) {
        super(x, y); // Si no se especifica, es: super();
        this.setT(t);
    }

    public int getT() {
        return this.t;
    }
    public final void setT(int t) {
        this.t = t;
    }
    public double velocidad() {
        return this.modulo() / this.getT();
    }
}

```

Cuando creamos una instancia de *PuntoTiempo*, se crea una instancia que lleva la funcionalidad de *Object*, *Punto* y lo añadido en *PuntoTiempo*. En la clase *PuntoTiempo*, en el constructor sin parámetros, como no hemos especificado el constructor de la superclase, se llama al *super()*.

## COMPATIBILIDAD ENTRE VARIABLES DE CLASES

Una variable de tipo *MiClase*, puede contener instancias de *MiClase* o de cualquier subclase. En definitiva, una variable de tipo *Object*, puede contener cualquier instancia. En cambio, está prohibido que una variable tenga instancias de las superclases.

En el ejemplo anterior, una variable de tipo *Punto*, puede referencia instancias de tipo *PuntoTiempo*, pero una variable de tipo *PuntoTiempo* **NO** puede tener instancia de la clase *Punto*.

También se puede realizar casting entre clases. Mostremos un ejemplo con código Java:

```

Punto p = new PuntoTiempo(1,2,3);
//Error: PuntoTiempo pt = new Punto(3,2);
p.setX(3);
//Error con p.setT(3);
((PuntoTiempo)p).setT(3); // Error con (PuntoTiempo)p.setT(3);

```

## REDEFINICIÓN DE MÉTODOS

A veces, los métodos heredados de la superclase no son adecuados para la subclase. En este caso, debemos redefinir (sobrescribir) el método.

Supongamos que queremos realizar una clase *Punto3D* que hereda de *PuntoTiempo* para añadirle una tercera dimensión. En este caso, el método heredado *modulo* ya no es válido y lo debemos redefinir. Es recomendable añadir la directiva *@Override* para asegurarnos que no realizamos una sobrecarga por error. A continuación mostramos el código Java.

#### Punto3D.java

```
public class Punto3D extends PuntoTiempo {
    private int z;
    public Punto3D(int x, int y, int z, int t) {
        super(x, y, t);
        this.setZ(z);
    }

    public int getZ() {
        return this.z;
    }
    public final void setZ(int z) {
        this.z = z;
    }

    @Override
    public double modulo() {
        return Math.sqrt(this.getX() * this.getX() +
            this.getY() * this.getY() + this.z * this.z);
    }
}
```

Fijarse que si enviamos un mensaje al método *velocidad* de una instancia de la clase *Punto3D*, esta se resuelve por la herencia de *PuntoTiempo*, cuando desde esa instancia se invoca al método *modulo*, se lanza la nueva versión redefinida en *Punto3D*.

## CLASE OBJECT

Esta clase es la superclase de todas. Algunos de sus métodos son:

- *public String toString()*. Este método devuelve una cadena con la descripción de la clase. Es importante redefinir este método para adaptarlo a nuestra descripción, es muy útil en los procesos de depuración. Por ejemplo, en la clase *Punto* podría devolver los valores *x* e *y* entre paréntesis y separados por una coma: [3,5]. Se recomienda que siempre se redefina este método. Su código quedaría:

```
@Override
public String toString() {
    return "[" + this.getX() + "," + this.getY() + "]";
}
```

Otra posible solución sería:

```
@Override
public String toString() {
    return "Punto[" + this.getX() + "," + this.getY() + "]";
}
```

También, esta es una alternativa:

```
@Override
public String toString() {
    return "Punto[x:" + this.getX() + ",y:" + this.getY() + "]";
}
```

- *public boolean equals(Object obj)*. Este método devuelve un boolean indicando si la instancia actual es igual a la que se pasa por parámetro. En *Object* compara direcciones, devolviendo true si son la misma dirección. Normalmente, esta no es la implementación necesitada. Por ejemplo, en la clase *Punto* debemos considerar que dos instancias de punto son iguales si la coordenada *x* e *y* coinciden. Su código quedaría:

```
@Override
public boolean equals(Object obj) {
    boolean result;
    if (obj == null || this.getClass() != obj.getClass()) {
        result = false;
    } else {
        final Punto otro = (Punto) obj;
        result = otro.getPtX() == this.ptX;
    }
    return result;
}
```

}

- `public int hashCode()`. Devuelve un código hash de este objeto. Normalmente, si se sobrescribe `equals` se debe sobre escribir `hashCode`. Se debe buscar que el código hash sea diferente entre dos instancias que no se consideren iguales.
- `protected void finalize() throws Throwable`. Este es el método destructor, se debe redefinir si queremos realizar alguna acción para liberar recursos. Cuando un objeto queda huérfano, el recolector de basura lo destruye; pero esta acción no es inmediata. Si queremos que actúe el recolector, lo podemos lanzar mediante la clase `System`: `System.gc()`. Un ejemplo sería:

```
@Override
protected void finalize() throws Throwable {
    System.out.println("Se liberan recursos");
    super.finalize();
}
```

- `protected Object clone() throws CloneNotSupportedException`. Este método nos proporciona una copia de la instancia, pero para ello, la clase a duplicar debe implementar el interface `Cloneable` (no tiene ningún método). Nuestra clase clone, debe encargarse de duplicar todas las instancias que lo formen. En un apartado posterior, veremos un ejemplo completo.

## FINAL

Si lo aplicamos a una clase, no se permite que existan subclases.

Si lo aplicamos a un método, no se permite que subclases redefinan el método.

## Clases abstractas

Una clase abstracta es una clase que no se puede instanciar, pero si heredar. También, se pueden definir constructores. Se utilizan para englobar clases de diferentes tipos, pero con aspectos comunes. Se pueden definir métodos sin implementar y obligar a las subclases a implementarlos.

Por ejemplo, podemos tener una clase *Figura*, que representa una figura general, y subclases con figuras concretas (*Triangulo*, *Circulo*...). Podemos establecer métodos comunes como *dibujar*, que sin conocer el tipo de figura, sea capaz de dibujarla. Pero para que esto funcione correctamente, cada figura concreta debe implementar su versión particular de *dibujar*.

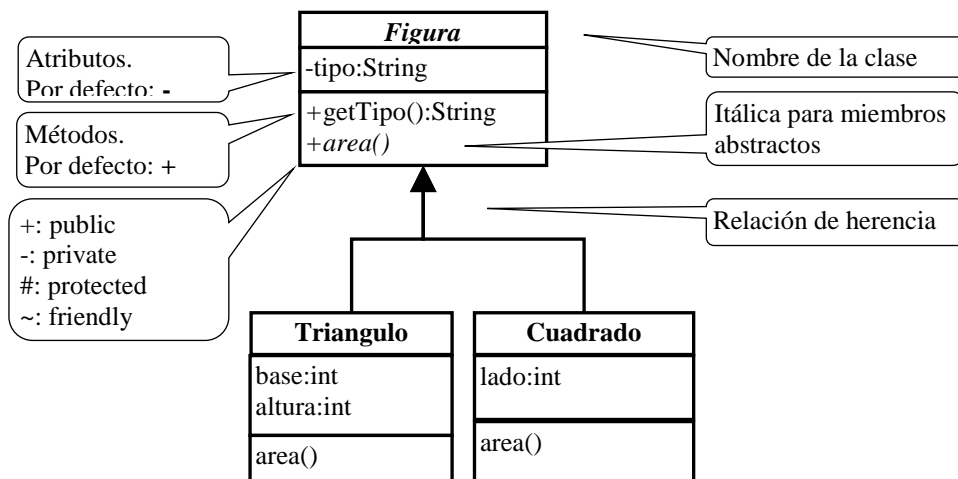
Si añadimos el calificador *abstract* después del calificativo *public* del método, le convertimos en abstracto y no hace falta que lo implementemos, basta con definirlo. Si añadimos *abstract* después del calificativo *public* de la clase, la convertimos en clase abstracta; esta acción es obligatoria si algún método es abstracto.

La ventaja en utilizar clases abstractas, es que podemos trabajar con variables o parámetros de tipo *Figura* y llamar a los métodos comunes sin saber a priori el tipo concreto de *Figura*. Esto permite en el futuro, añadir nuevas *Figuras*, sin cambiar las clases ya desarrolladas. A este concepto se le llama polimorfismo.

El polimorfismo es el modo en que la POO implementa la polisemia, es decir, un solo nombre para muchos conceptos.

Este tipo de polimorfismo se debe resolver en tiempo de ejecución y por ello hablamos de polimorfismo dinámico, a diferencia del polimorfismo estático que se puede resolver en tiempo de compilación. Recordemos que éste último se implementaba mediante la sobrecarga de los métodos.

A continuación presentamos un ejemplo mediante notación UML (Lenguaje Unificado de Modelado) y en Java



#### Figura.java

```

public abstract class Figura {
    private String tipo;
    public Figura(String tipo) {
        this.tipo = tipo;
    }
    // getters & setters
    public abstract double area();
}
  
```

#### Triangulo.java

```

public class Triangulo extends Figura {
    private int base, altura;
    public Triangulo(String tipo, int base, int altura) {
        super(tipo);
        this.setBase(base);
        this.setAltura(altura);
    }
    // getters & setters
    @Override
    public double area() {
        return (double) this.base * this.altura / 2;
    }
}
  
```

#### Cuadrado.java

```

public class Cuadrado extends Figura {
    private int lado;
    public Cuadrado(String tipo, int lado) {
        super(tipo);
        this.setLado(lado);
    }
    // getters & setters
    @Override
    public double area() {
        return (double) this.lado * this.lado;
    }
}
  
```

#### Polimorfismo.java

```

...
public void polimorfismo(Figura una) {
    System.out.println("Tipo: " + una.getTipo());
    System.out.println("Area: " + una.area());
}
...
  
```

 Crear las clases *Figura*, *Triangulo* y *Cuadrado*

## Interfaces

Un interface en Java es un tipo abstracto que representa un conjunto de métodos que las clases pueden implementar y un conjunto de constantes estáticas, es similar a una clase abstracta pura. Igual que las clases abstractas, los interfaces no se puede instanciar. Su utilidad es crear variables de tipo interface e independizarse de las clases que contiene.

Para definir un interface se realiza de forma similar a una clase abstracta pero con la palabra *interface* en lugar de *class*. Pueden existir variables o parámetros de tipo interface, pero sólo pueden contener instancias de clases que hayan implementado el interface.

Una clase puede implementar uno o varios interfaces, para ello se utiliza la palabra *implements*.

Veamos un ejemplo.

### IBox.java

```
public interface IBox {  
    public void put(Object objeto);  
    public Object get();  
}
```

### CBox.java

```
public class CBox implements IBox {  
    private Object objeto;  
  
    public CBox() {  
        this.objeto=null;  
    }  
  
    @Override  
    public Object get() {  
        return this.objeto;  
    }  
  
    @Override  
    public void put(Object objeto) {  
        this.objeto = objeto;  
    }  
}
```

Los interfaces pueden heredar de otros mediante la palabra *extends*, además se permite herencia múltiple.

Mediante interfaces también se aporta el polimorfismo, de hecho, una clase abstracta con todos sus métodos abstractos es similar a un interface.

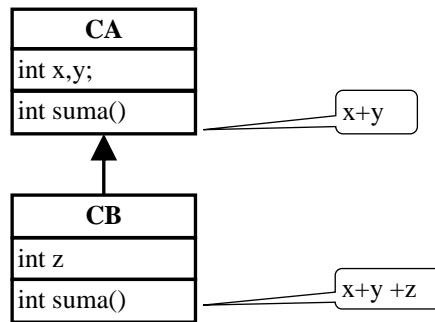
En general, es mejor utilizar interface; sólo cuando se necesite añadir código, se debe utilizar las clases abstractas.

## COMPATIBILIDAD DE VARIABLES

Una variable de tipo interface puede contener instancias de cualquier clase que implemente el interface.

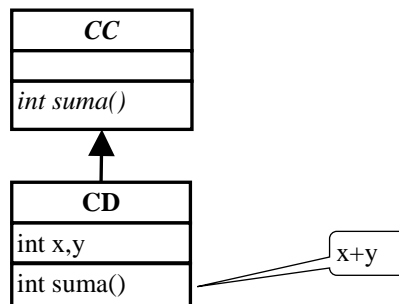
## Ejercicios

 Implementar las clases CA y CB, donde CA hereda de CB



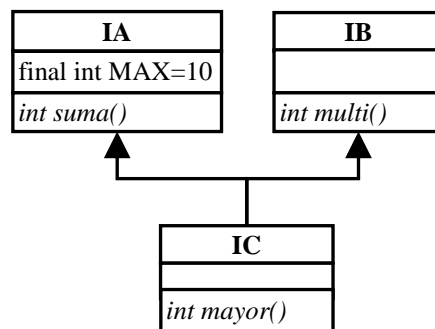
✍ Dado el código: `CA a = new CB(1,2,3);` ¿qué valor devuelve “`a.suma();`”?

✍ Implementar las clases `CC` y `CD`



✍ Añadir e implementar el método `int dobleSuma()` en `CC`

✍ Definir el interface `IA`, `IB` e `IC`



✍ Ampliar `CA` para que implemente `IC`

✍ interface `Comparable`, método `compareTo(Object uno, Object dos)` (si `uno` es `<`, `==` o `>` que `dos`, devuelve `-1`, `0`, `+1` respectivamente)

✍ Ampliar `CA` para que implemente `Comparable`

## Colaboración entre clases

Llegados a este punto, se recomienda pasar a un entorno de desarrollo avanzado, como por ejemplo Eclipse. En el área de documentación del MIW se encuentra un tutorial rápido sobre Eclipse.



## Relación de clases

Una aplicación orientada a objetos es una colección de objetos, algunos jerárquicos, que colaboran entre sí para dar una funcionalidad global. Cada objeto debe asumir una parte de las responsabilidades y delegar otras a los objetos colaboradores.

Existirá un objeto que asume las funciones de la aplicación principal, será el encargado de crear las instancias oportunas y tener un único punto de entrada (*main*).

Cabe hacernos una pregunta fundamental, ¿en cuántos objetos debemos descomponer la aplicación? Si realizamos objetos muy pequeños, cada uno de ellos será fácil de desarrollar y mantener, pero resultará difícil la integración. Si realizamos objetos grandes, la integración será fácil, pero el desarrollo de los objetos complicado. Se debe buscar un equilibrio.

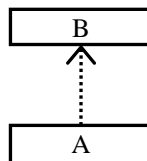
En definitiva, y sin adentrar más en la estructuración de las clases de una aplicación, nuestra aplicación tendrá un conjunto de clases que están relacionadas entre ellas.

Se considera que existe una **relación entre clases** si sus objetos colaboran, es decir, se mandan mensajes. Tenemos, principalmente, tres tipos de relaciones entre clases: uso, agregación y composición, aparte de la herencia que ya vimos.

### RELACIÓN DE USO

Se dice que existe una relación de uso entre la clase A y la clase B, si un objeto de la clase A lanza mensajes al objeto de la clase B, para utilizar sus servicios.

Es una relación que se establece entre un cliente y un servidor, en un instante dado, a través de algún método. No es una relación duradera, sino esporádica.

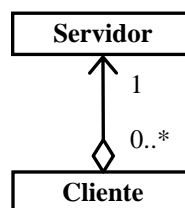


```
public class A {
    ...
    public void metodo(B b) {
        b.servicio();
    }
    ...
}
```

### RELACIÓN DE AGREGACIÓN

Se dice que la clase A tiene una relación de agregación con la clase B, si un objeto de la clase A delega funcionalidad en los objetos de la clase B.

El ciclo de vida de ambos no coincide. La clase A delega parte de su funcionalidad, pero no crea el objeto B. Varios objetos pueden estar asociados con el objeto A. Al objeto A, le deben pasar la referencia del objeto B. Es una relación duradera entre cliente (A) y servidor (B), en la vida del cliente.



```
public class Cliente {
    private Servidor servidor;
    public Cliente(Servidor servidor) {
        this.servidor = servidor;
    }
}
```

```

    }
    ...
}

```

```

public class Servidor {
    ...
    public String servicio() {
        return "Servido";
    }
    ...
}

```

## RELACIÓN DE COMPOSICIÓN

Se dice que existe una relación de composición entre la clase A y la clase B, si la clase A contiene un objeto de la clase B, y delega parte de sus funciones en la clase B.

Los ciclos de vida de los objetos de A y B coinciden. El objeto A es el encargado de crear la instancia de B y de almacenarla, siendo su visibilidad privada. Cuando se destruye A, también se destruye B.

Por ejemplo, se pretende realizar una clase llamada Agenda y para ello se apoya en la clase Lista, es decir, dentro de la clase Agenda se crea una instancia de la clase Lista, y algunas de las funcionalidades que aporta la clase Agenda se realizan con los métodos de la clase Lista.



```

public class Agenda {
    private Lista lista;

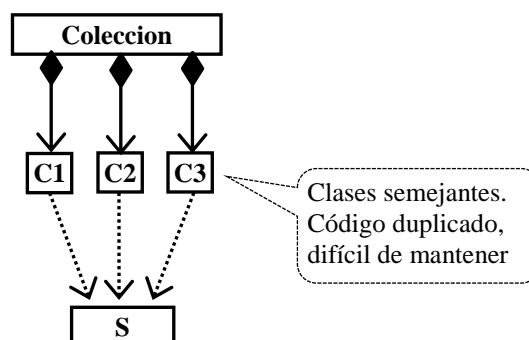
    public Agenda() {
        this.setLista(new Lista());
    }
    ...
}

```

## Beneficio de la herencia

Es importante realizar un uso de la herencia para optimizar el código y evitar las redundancias.

**Ejemplo incorrecto.** Veamos un mal uso de las clases, donde aparece código redundante y un mal uso de la herencia. Resulta incorrecto que el mensaje lo interprete el emisor



```

//MAL DISEÑO
public class Coleccion{
    ...
    for(C item: miColeccion){

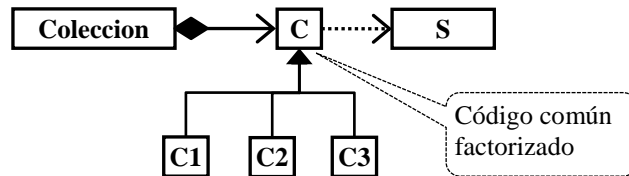
```

```

if (item.getName.equals("C1"){...procesar C1}
else if (item.getName.equals("C2"){...procesar C2}
...
}

```

**Ejemplo correcto.** Buen uso de la herencia. El mensaje lo interprete el receptor



```

//BUEN DISEÑO
public class Coleccion{
...
for(C item: miColeccion){
    item.procesar();
}
}

```

## Beneficio de los interfaces

En una relación de agregación entre la clase A y la clase B, la clase A debe lanzar mensajes a la clase B, y para ello es necesario que a la clase A le pasen la referencia de la clase B.

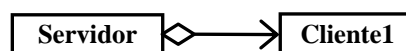
Una vez implementada la clase A, cualquier cambio en la clase B (por ejemplo, una mejora o cambio en la implementación de B) provoca que debiéramos cambiar todas las referencias de la clase A sobre la B. Incluso puede que cuando se desarrolla la clase A, no se dispone de la implementación de la clase B.

Para ello disponemos de dos mecanismos:

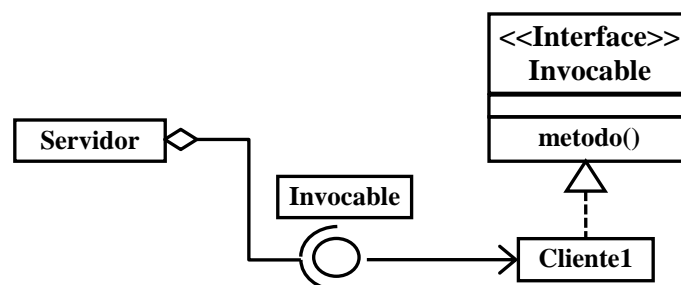
- Mediante herencia de clases. En este caso, la clase A referencia los métodos de una superclase B, pudiendo existir varias subclases de B que sirvan a tal fin.
- Mediante interfaces. Aquí, la clase A referencia los métodos de un interface, pudiendo existir un conjunto de clases que implementan ese interface.

Fijarse, que en ambos casos, cualquier cambio en las implementaciones de B no afectan al código de la clase A.

Veamos un ejemplo concreto. En esta asociación existe una dependencia clara entre la clase *Servidor* y la clase *Cliente*.



Pero en este ejemplo tenemos un problema, ya que se pretende construir un servidor para que de servicio a muchos clientes y se debe romper esa dependencia. Obligar a los clientes a heredar de una clase no es una buena solución, ya que le limitamos en su diseño. La mejor solución es utilizar los interfaces. A continuación presentamos el nuevo diseño:



En este nuevo diseño, hemos roto la dependencia del servidor respecto a las diferentes implementaciones de los clientes.

Veamos el código en Java:

#### Invocable.java

```
public interface Invocable {  
    public void newMsg(String msg);  
}
```

#### Servidor.java

```
public class Servidor {  
    private Invocable cliente;  
  
    public void add(Invocable cliente) {  
        this.cliente = cliente;  
    }  
  
    public void remove() {  
        this.cliente = null;  
    }  
  
    public void broadcast() {  
        if (this.cliente != null) cliente.newMsg("Nuevo mensaje");  
    }  
}
```

#### Cliente1.java

```
public class Cliente1 implements Invocable {  
    public Cliente1(Servidor servidor) {  
        servidor.add(this);  
    }  
  
    @Override  
    public void newMsg(String msg) {  
        System.out.println("Llega un nuevo mensaje..." + msg);  
    }  
}
```

## Paquetes

Los paquetes agrupan un conjunto de clases que trabajan conjuntamente sobre el mismo ámbito. Es una facilidad ofrecida por Java para agrupar sintácticamente clases que van juntas conceptualmente y definir un alto nivel de protección para los atributos y los métodos.

La ventaja del uso de paquetes es que las clases quedan ordenadas y no hay colisión de nombres. Si dos programadores llaman igual a sus clases y luego hay que juntar el código de ambos, basta explicitar a qué paquete nos referimos en cada caso.

La forma de nombrar los paquetes es con palabras (normalmente en minúsculas) separadas por puntos, estableciendo una jerarquía de paquetes; la jerarquía de paquetes es independiente de la jerarquía de clases. Las clases deben almacenarse en una estructura de carpetas que coincidan con la estructura de paquetes.

Para que una clase se asocie a un paquete se realiza con la sentencia *package*, se debe situar antes de la clase. Algunos ejemplos de uso son:

```
public class Prueba ...  
  
package p1.p2;  
public class MiClase ...  
  
package p1;  
public class Clase2 ...
```

Tanto los ficheros *\*.java* como los ficheros *\*.class*, se deben almacenar en una jerarquía de carpetas que coincidan con los nombres de los paquetes. En la Figura 4. Paquetes en JavaFigura 4 se muestra cómo quedaría la jerarquía de carpetas.

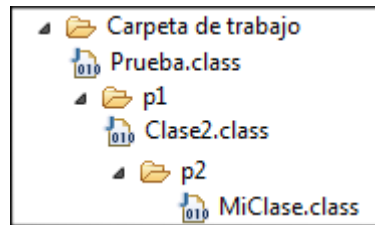


Figura 4. Paquetes en Java

Para su distribución, se genera un fichero *jar*, con la estructura de carpetas y clases. Recordar que este fichero no hace falta descomprimirlo. Para su utilización, se debe referenciar mediante la variable de entorno *CLASSPATH*.

Para referenciar cualquier clase en Java, se debe poner el paquete al que pertenece. Por ejemplo: *java.lang.String*, *java.lang.Integer*... Hasta ahora no ha sido necesario, porque el paquete *java.lang* se importa de forma automática.

Para evitar poner siempre la jerarquía de paquetes, se puede utilizar la sentencia *import*, con ello se puede importar un paquete entero (pero no los paquetes inferiores) o una clase.

Mostramos un ejemplo de uso:

```
public class Una{
    private java.util.Map mapa;

    public Una() {
        mapa = new java.util.HashMap();
        mapa.put("clave", "valor");
    }
    // ...
}
```

```
import java.util.Map;
import java.util.*; //importa todas las clases del paquete, NO los subpaquetes.
public class Una{
    private Map mapa;
    // ...
}
```

Los paquetes permiten utilizar los modos de protección *protected* y *friendly*, que veremos en el siguiente apartado.

**Nota.** Se recomienda utilizar un IDE, como por ejemplo *Eclipse*. Para ejecutar una clase de un paquete desde la consola que no está referenciado mediante la variable de entorno *CLASSPATH*, nos debemos situar en la carpeta raíz de los paquetes y referenciar al fichero *class* mediante la ruta completa.

Un ejemplo es: `java paquete/MiClase`

## Visibilidad

Hasta ahora hemos trabajado con dos tipos de visibilidad: *public* y *private*. Recordemos que un elemento público es visible para todos y privado sólo para uso interno.

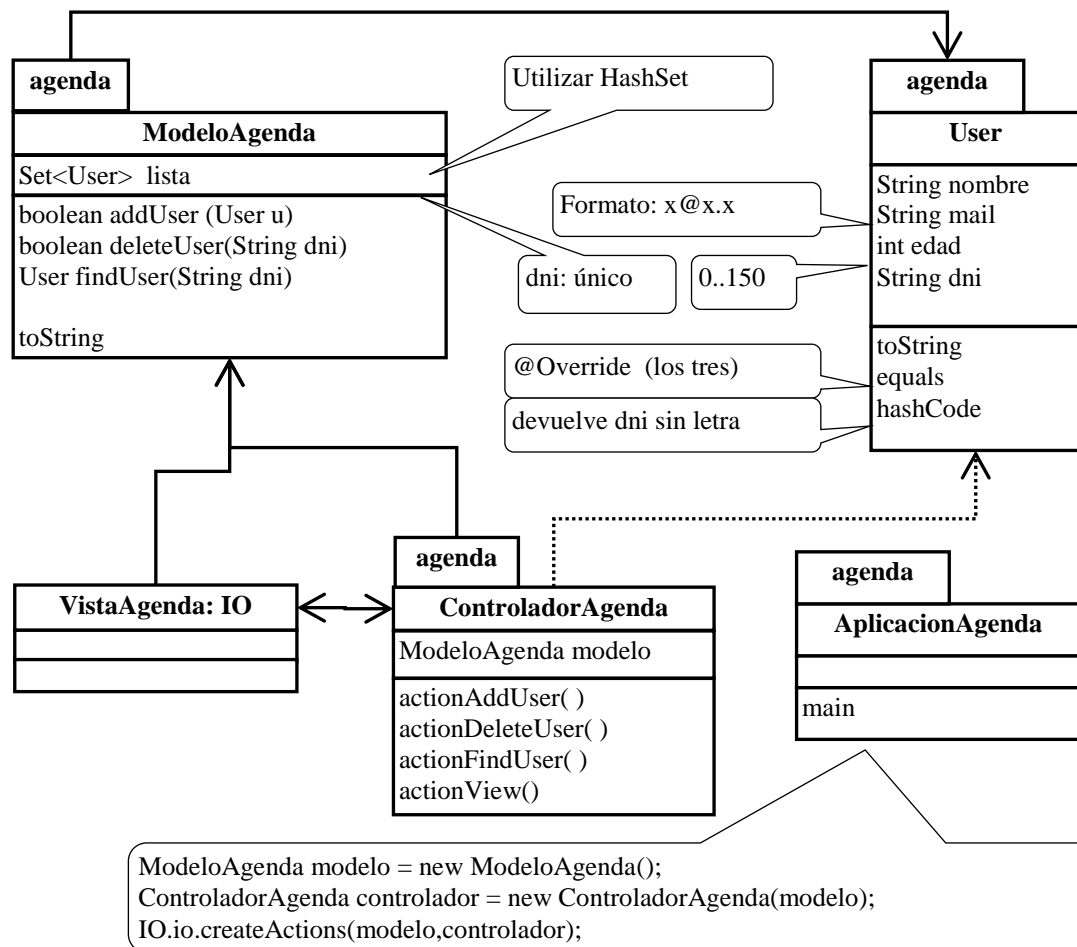
En Java se distinguen dos tipos más de visibilidad: *protected* y sin calificador (*friendly*).

Cuando calificamos a un elemento como *protected*, significa que es visible dentro del mismo paquete o cualquier subclase de forma directa (mediante *super.elemento*). Si pretendemos que una subclase acceda de forma directa a los atributos de la superclase, el calificador apropiado es *protected*.

Cuando dejamos a un elemento sin calificar (*friendly*), significa que sólo es visible dentro del mismo paquete.

 Probar las clases anteriores

**AVANZADO. Implementar la aplicación AGENDA. Lleva una agenda de usuarios. Permite las acciones básicas: añadir, borrar, buscar y ver toda la agenda.**



## Método de Desarrollo de Programas

Un programa orientado a objetos en ejecución es una colección jerárquica de objetos contenidos en otros objetos sucesivas veces que colaboran todos entre sí. Paralelamente, un programa en edición/compilación es una colección de las clases de estos objetos relacionadas entre sí.

Las funciones del **Programa Principal** las asumirá un único objeto que englobe todo el sistema de objetos (mediante un desencadenamiento de instanciaciones) al que lanzándole un único mensaje realice todas las funciones deseadas (mediante un desencadenamiento de mensajes). Fuera de ese objeto no puede existir nada, no tiene sentido que reciba parámetros, no tiene sentido devolver un valor a nadie.

La clase de este objeto **Principal** suele responder a nombres como Aplicación, Sistema, Gestor, Simulador, Juego, etc. El mensaje a **Principal** suele responder a nombres como ejecutar, gestionar, simular, jugar, etc. Por tanto, de forma genérica para cualquier programa orientado a objetos:

```

Cliente1.java
class Aplicacion {
    // ...
    public void ejecutar() {
        // ...
    }
    public static void main(String[] arg) {
        Aplicacion aplicacion = new Aplicacion();
        aplicacion.ejecutar();
    }
}
  
```

```
}
```

Una vez establecido el objeto *Principal* e inicial queda por determinar cuántos otros objetos existen en el programa: ¿hasta qué punto se realizan descomposiciones sucesiva de objetos en objetos que forman la jerarquía de composición, fundamental en un programa orientado a objetos? Dado el mismo conjunto de requisitos, más módulos significan menor tamaño individual del módulo. Sin embargo, a medida que crece el número de módulos, el esfuerzo asociado con la integración de módulos también crece. Estas características llevan a una curva de coste total. Hay un número, M, de módulos que resultaría en un coste de desarrollo mínimo, pero no tenemos la sofisticación necesaria para predecir M con seguridad.

Entonces, dado que no se puede establecer el número de distintas clases que tiene un programa, se recurre a la siguiente regla de desarrollo: aceptar una clase más si existen fundamentos para pensar que se reducen los costes de desarrollo totales que favorezca mayores cotas de abstracción, encapsulación, modularización y jerarquización que faciliten la legibilidad, mantenimiento, corrección, etc.

Una vez que se conoce de dónde partir (la clase del objeto *Principal*) y hasta dónde llegar (hasta que nuevas clases no reduzcan los costes), resta saber qué camino seguir para el desarrollo del programa.

El método propuesto es una simplificación extrema y variación de HOOD (diseño orientado a objetos jerárquico). El resultado es un método muy sencillo y utilizable en pequeñas aplicaciones. Este método contiene 5 fases que se aplican cíclicamente para cada clase del programa:

1. Tomar una clase
2. Definir el comportamiento
3. Definir los atributos
4. Codificar los métodos
5. Recapitular nuevas clases y métodos.

A continuación se aplicará el método fase a fase bajo la siguiente nomenclatura: "F.C" siendo 'F' la fase actual (con valores de 1 a 5) y 'C' el ciclo de aplicación (con valores de 1 a N clases del programa). Posteriormente, se sintetizarán esquemáticamente todas las fases del método.

El programa a resolver debe permitir a dos usuarios jugar a las "Tres En Raya" (TER):

- Inicialmente el tablero estará vacío
- Los jugadores pondrán las fichas alternativamente, siendo posible alcanzar las "Tres en Raya" con la quinta puesta
- En caso contrario, una vez puestas las 6 fichas, se moverán las fichas alternativamente
- Tanto en las puestas como en los movimientos, se deben requerir las coordenadas a usuario validando la corrección de la puesta/ movimiento
- Se debe detectar automáticamente la finalización del juego al lograrse "Tres en Raya": tras la quinta y sexta puesta y tras cada movimiento

(1.1) *Tomar una clase*: al principio de todo el desarrollo del programa será la clase del objeto que engloba todo el sistema: *TER*.

```
class TER {
    public static void main(String[] arg) {
        TER partida = new TER();
        partida.jugar();
    }
}
```

(2.1) *Determinar el comportamiento*: al principio de todo el desarrollo del programa será el método de la clase que engloba todo el sistema que dispare toda la funcionalidad deseada -jugar-: no tendrá parámetros y no devolverá nada; el juego es "todo" el universo de objetos.

```
class TER {
    public void jugar() {
        // ..
    }
    ...
}
```

**(3.1) Determinar los atributos con sus respectivas clases;** no es estrictamente necesario determinar todos y cada uno de los atributos, sino que se deben establecer aquellos que sean “obvios” en el momento; si se omitiese alguno surgirá su necesidad en el siguiente punto (4.1).

Los atributos soportarán los datos que posibiliten el proceso del juego: memorizar dónde están las fichas puestas/movidas, saber a qué jugador le corresponde poner/mover y su color de fichas, etc.

En programación orientada a objetos, se recurre a atributos que son objetos de otras clases responsables de los detalles del soporte y manipulación de esos datos;

Para el programa de las "Tres en Raya":

- Se crea un objeto de la clase Tablero; responsable de aparecer vacío, guardar las fichas que se pongan, detectar las "Tres en Raya", mostrar la situación de las fichas, etc.
- Se crean dos objetos de la clase Jugador; responsables de memorizar el color de cada uno, comunicarse con cada usuario y controlar automáticamente sus puestas y movimientos, etc.
- Se crea un objeto de la clase Turno; responsable de memorizar a qué jugador le corresponde poner/mover, alternar el jugador, etc.

```
public class TER {
    private final Tablero tablero = new Tablero();
    private final Jugador[] jugadores = new Jugador[2];
    private final Turno turno = new Turno();

    public TER() {
        jugadores[0] = new Jugador('o');
        jugadores[1] = new Jugador('x');
    }

    public void jugar() {
        // ...
    }

    public static void main(String[] arg) {
        TER partida = new TER();
        partida.jugar();
    }
}
```

**(4.1) Codificar métodos:** no debe ser un impedimento incluir en la codificación de los métodos el paso de mensajes correspondientes a métodos que no se encuentren definidos previamente en sus correspondientes clases. Hay que codificar los métodos de la clase actual, TER, delegando tareas a los objetos de las clases de los atributos, Tablero, Jugador y Turno.

```
public void jugar() {
    tablero.mostrar();
    for (int i = 0; i < 5; i++) {
        jugadores[turno.toca()].poner(tablero);
        turno.cambiar();
        tablero.mostrar();
    }
    if (tablero.hayTER()) {
        jugadores[turno.noToca()].victoria();
    } else {
        jugadores[turno.toca()].poner(tablero);
        tablero.mostrar();
        while (!tablero.hayTER()) {
            turno.cambiar();
            jugadores[turno.toca()].mover(tablero);
            tablero.mostrar();
        }
        jugadores[turno.toca()].victoria();
    }
}
```

**(5.1) Recapitular:** nuevas clases aparecidas en los puntos (2.1), (3.1) y (4.1) y métodos correspondientes a los mensajes enviados a objetos de estas clases en el punto (4.1). El valor devuelto y parámetros de cada método se extraen del contexto del envío del mensaje.

La recapitulación abre de nuevo la aplicación de las 5 fases del método para cada una de las clases, pero con los siguientes matices:



- La primera fase, tomar una clase, que "anulada" al limitarse a escoger una de las clases recapituladas anteriormente. Se recomienda que la elección sea la que inspire más riesgos en su implantación
- La segunda fase, determinar el comportamiento público de clases anteriores queda "anulada" por la quinta fase de recapitulación anterior

(1.2) *Tomar una clase: Jugador* de la fase (5.1).

(2.2) *Determinar el comportamiento:* de la fase (5.1).

(3.2) *Determinar los atributos con sus respectivas clases;* para poner, mover y cantar victoria un Jugador solo precisa saber el color de sus fichas.

```
public class Jugador {
    private char color;
    public Jugador(char c) {
        // TODO Auto-generated constructor stub
    }
    public void poner(Tablero tablero) {
        // TODO Auto-generated method stub
    }

    public void victoria() {
        // TODO Auto-generated method stub
    }

    public void mover(Tablero tablero) {
        // TODO Auto-generated method stub
    }
}
```

(4.2) *Codificar métodos:* no debe ser un impedimento incluir en la codificación de los métodos nuevas clases "auxiliares" no previstas hasta el momento.

```
public class Jugador {
    private char color;
    public Jugador(char color) {
        this.color = color;
    }
    public void poner(Tablero tablero) {
        IO.out.println("juega: " + color);
        Coordinada destino = new Coordinada();
        do {
            destino.recoger("Coordinada destino de puesta");
        } while (!destino.valida() || tablero.ocupado(destino));
        tablero.poner(destino, color);
    }
    public void mover(Tablero tablero) {
        IO.out.println("juega: " + color);
        Coordinada origen = new Coordinada();
        do {
            origen.recoger("Coordinada origen de movimiento");
        } while (!origen.valida() || tablero.vacio(origen));
        tablero.sacar(origen);
        tablero.mostrar();
        this.poner(tablero);
    }
    public void victoria() {
        IO.out.println("las " + color + " han ganado y ...");
    }
}
```

Esta solución opta por la inclusión de una nueva clase Coordinada que aglutine el tratamiento de la fila y la columna de una casilla del tablero donde poner/mover fichas.

(1.3) *Tomar una clase: Tablero* de la fase (5.1)

(2.3) *Determinar el comportamiento:* de las fases (5.1) y (5.2)

(3.3) *Determinar los atributos con sus respectivas clases;* para poner, sacar y detectar si *hayTER* o si una casilla está ocupado o vacía un Tablero solo precisa soportar 3x3 caracteres que memoricen las posiciones de las fichas (el carácter '-' representa la ausencia de fichas).

(4.3) *Codificar métodos.*

```

public class Tablero {
    private final char[][] fichas = new char[3][3];
    private static final char VACIO = '-';
    public Tablero() {
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                fichas[i][j] = VACIO;
            }
        }
    }
    public void mostrar() {
        IO.out.clear();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                IO.out.print(fichas[i][j]);
                IO.out.print(' ');
            }
            IO.out.println();
        }
        IO.out.println();
    }
    public boolean hayTER() {
        return this.hayTER('o') || this.hayTER('x');
    }
    private boolean hayTER(char color) {
        boolean victoria = false;
        int diagonal = 0;
        int inversa = 0;
        int[] filas = new int[3];
        int[] columnas = new int[3];

        for (int i = 0; i < 3; i++) {
            filas[i] = 0;
            columnas[i] = 0;
            for (int j = 0; j < 3; j++) {
                if (fichas[i][j] == color) {
                    if (i == j) {
                        diagonal++;
                    }
                    if (i + j == 2) {
                        inversa++;
                    }
                    filas[i]++;
                    columnas[j]++;
                }
            }
        }
        if ((diagonal == 3) || (inversa == 3)) {
            victoria = true;
        } else {
            for (int i = 0; i < 3; i++) {
                if ((columnas[i] == 3) || (filas[i] == 3)) {
                    victoria = true;
                }
            }
        }
        return victoria;
    }
    public boolean ocupado(Coordenada coordenada) {
        return fichas[coordenada.getFila()][coordenada.getColumna()] != VACIO;
    }
    public boolean vacio(Coordenada coordenada) {
        return (!this.ocupado(coordenada));
    }
    public void poner(Coordenada coordenada, char color) {
        fichas[coordenada.getFila()][coordenada.getColumna()] = color;
    }
    public void sacar(Coordenada coordenada) {
        this.poner(coordenada, VACIO);
    }
}

```

### (5.3) Recapitular

(1.4) Tomar una clase: Coordenada de la fase (5.2).

(2.4) *Determinar el comportamiento:* de las fases (5.2) y (5.3).

(3.4) *Determinar los atributos con sus respectivas clases;* para recoger, validar y obtener fila y columna de una coordenada, solo se precisa memorizar la fila y la columna.

(4.3) *Codificar métodos:*

```
public class Coordenada {
    private int fila;
    private int columna;

    public void recoger(String titulo) {
        fila = IO.in.readInt("Dame una fila");
        columna = IO.in.readInt("Dame una columna");
    }

    public boolean valida() {
        return fila <= 2 && fila >= 0 && columna <= 2 && columna >= 0;
    }

    public int getFila() {
        return fila;
    }

    public int getColumna() {
        return columna;
    }
}
```

(5.4) *Recapitular:* no surgen ni nuevas clases ni nuevos métodos de clases ya recapituladas.

(1.5) *Tomar una clase:* Turno de la fase (5.1)

(2.5) *Determinar el comportamiento:* de las fases (5.1)

(3.5) *Determinar los atributos con sus respectivas clases;* para cambiar y obtener a quien toca y el contrario solo se precisa memorizar un valor alternativo entre 1 y 2.

(4.5) *Codificar métodos:*

```
public class Turno {
    private int valor = 0;

    public void cambiar() {
        this.valor = (this.valor + 1) % 2;
    }

    public int toca() {
        return valor;
    }

    public int noToca() {
        return (valor + 1) % 2;
    }
}
```