

## Sumario

UT 06: Desarrollo de clases.....	3
1 Concepto de clase.....	3
2 Estructura y miembros de una clase.....	3
2.1 Declaración de una clase.....	3
2.2 Cabecera de una clase.....	4
2.3 Cuerpo de una clase.....	5
2.4 Miembros estáticos o de clase.....	6
3 Atributos.....	7
3.1 Declaración de atributos.....	8
3.2 Modificadores de acceso.....	9
3.3 Modificadores de contenido.....	10
3.4 Atributos estáticos.....	11
4 Métodos.....	13
4.1 Declaración de un método.....	13
4.2 Cabecera de un método.....	14
4.3 Modificadores en la declaración de un método.....	15
4.4 Parámetros en un método.....	16
4.5 Cuerpo de un método.....	17
4.6 Sobrecarga de métodos.....	21
4.7 Sobrecarga de operadores.....	22
4.8 La referencia this.....	23
4.9 Métodos estáticos.....	24
5 Encapsulación.....	25
5.1 Ocultación de atributos. Métodos de acceso.....	25
5.2 Ocultación de métodos.....	26
6 Utilización de métodos y atributos de una clase.....	29
6.1 Declaración de un objeto.....	29
6.2 Creación de un objeto.....	30
6.3 Manipulación de un objeto: utilización de métodos y atributos.....	31
6.4 Utilización de constructores.....	34
6.5 Constructores de copia.....	38
7 Introducción a la herencia.....	40
8 Empaquetado de clases.....	42
8.1 Jerarquía de paquetes.....	42
8.2 Utilización de los paquetes.....	43
8.3 Inclusión de una clase en un paquete.....	44
8.4 Proceso de creación de un paquete.....	45
9 Librerías de objetos.....	45
9.1 Sentencia import.....	46
9.2 Compilar y ejecutar clases con paquetes.....	46

---

9.3 Jerarquía de paquetes.....	47
9.4 Librerías de Java.....	47
10 Programación de consola: entrada y salida de información.....	49
10.1 Conceptos sobre la clase System.....	49
10.2 Entrada por teclado. Clase System.....	51
10.3 Entrada por teclado. Clase Scanner.....	52
10.4 Salida por pantalla.....	53
10.5 Salida de error.....	54
11 Documentación del código.....	54
11.1 Etiquetas y posición.....	55
11.2 Uso de las etiquetas.....	56
11.3 Orden de las etiquetas.....	57

## UT 06: Desarrollo de clases

### 1 Concepto de clase

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de las características de los objetos que forman parte de ella, indicando:

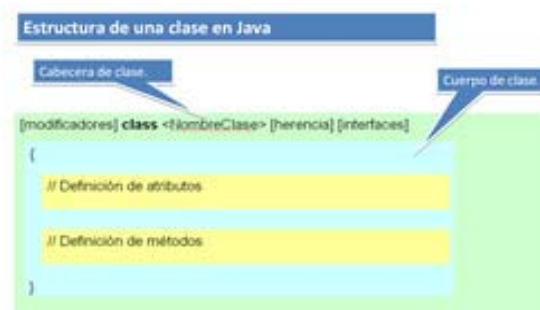
- Los atributos comunes a los objetos que pertenezcan a esa clase (información).
- Los métodos que permiten interactuar con esos objetos (comportamiento).

Podemos ver la clase como una plantilla o modelo para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar. Así pues, los objetos son instancias concretas de las clase, y estas serían una abstracción o definición de lo que clasificamos como “familias” de objetos.

### 2 Estructura y miembros de una clase

Los programas están formados por un conjunto de clases, a partir de las cuales se crean objetos que se relacionan unos con otros. Para declarar una clase en Java se usa la palabra reservada **class**. En la declaración de una clase aparecen los elementos:

- **Cabecera de la clase.** Formada por los modificadores de acceso, la palabra reservada **class** y el nombre de la clase.
- **Cuerpo de la clase.** En él se especifican los distintos miembros de la clase: atributos y métodos. Es decir, el contenido de la clase.



En el cuerpo de la clase se declaran los atributos que caracterizan a los objetos de la clase y se define e implementa el comportamiento de dichos objetos (se declaran e implementan los métodos).

#### 2.1 Declaración de una clase

La declaración de una clase en Java tiene la siguiente estructura general:

```
[modificadores] class <NombreClase> [herencia] [interfaces] { // Cabecera de la
clase
// Cuerpo de la clase
Declaración de los atributos
Declaración de los métodos
}
```

Un ejemplo básico podría ser:

```
/**
 *
 * Ejemplo de clase Punto
 */
class Punto {
// Atributos
int x,y;
// Métodos
int obtenerX () { return x; }
int obtenerY() {return y;}
void establecerX (int vx) { x= vx; };
void establecerY (int vy) { y= vy; };
}
```

En este caso se trata de una clase muy sencilla en la que el cuerpo de la clase (el área entre las llaves) contiene el código y declaraciones necesarias para que los objetos que se construyan (basándose en esta clase) puedan funcionar apropiadamente en un programa (declaraciones de atributos para contener el estado del objeto y métodos que implementen el comportamiento de la clase y los objetos creados a partir de ella).

Al implementar una clase Java es necesario tener en cuenta:

- Por convenio, los nombres de las clases empiezan por una letra mayúscula. Si el nombre de la clase está formado por varias palabras, cada una también comienza con mayúscula. Siguiendo esta recomendación, algunos ejemplos de nombres de clases podrían ser: Recta, Circulo, Coche, JugadorFutbol, AnimalMarino, etc.
- El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase para poder usarla desde otras clases que están fuera del archivo.
- Tanto la definición como la implementación de una clase se incluye en el mismo archivo (archivo ".java"). En otros lenguajes como por ejemplo C++, definición e implementación podrían ir en archivos separados (archivos .h y .cpp en C++).

## 2.2 Cabecera de una clase

En general, la declaración de una clase incluye estos elementos y en este orden:

1. Modificadores tales como public, abstract o final.
2. El nombre de la clase (la primera letra de cada palabra mayúscula, por convenio).
3. El nombre de su clase padre (superclase), si es que se especifica, precedido por la palabra reservada extends ("extiende" o "hereda de").
4. Una lista separada por comas de interfaces que son implementadas por la clase, precedida por la palabra reservada implements ("implementa").
5. El cuerpo de la clase, encerrado entre llaves {}.

La sintaxis completa de una cabecera (los cuatro primeros puntos) queda de la forma:

```
[modificadores] class <NombreClase> [extends <NombreSuperClase>] [implements  
<NombreInterface1>] [[implements<NombreInterface2>] ...] {  
...  
}
```

En el ejemplo anterior de la clase Punto teníamos la siguiente cabecera:

```
class Punto {
```

En este caso no hay modificadores, indicadores de herencia ni implementación de interfaces. Solo la palabra reservada class y el nombre. Es lo mínimo que puede haber en la cabecera de una clase.

La herencia y las interfaces se verán más adelante. Veamos los modificadores que se pueden indicar al crear la clase y qué efectos tienen. Los modificadores de clase son:

```
[public] [final | abstract]
```

Veamos qué significado tiene cada uno de ellos:

- Modificador **public**. Esta clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase (otra parte del programa). Si no se especifica "public" la clase sólo se puede usar desde clases en el mismo paquete (veremos este concepto más adelante). Sólo puede haber una clase public (clase principal) en un archivo .java. El resto de clases que se definan en ese archivo no serán públicas.
- Modificador **abstract**. Indica que la clase es abstracta. Una clase abstracta no es instanciable, es decir, no se pueden crear objetos de esa clase, solo de otras que hereden de ella. Este concepto de herencia, que veremos más adelante, puede resultar útil para crear una jerarquía de clases.
- Modificador **final**. Indica que no podrás crear clases que hereden de ella. También volveremos a este modificador cuando estudiemos el concepto de herencia. Los modificadores final y abstract son excluyentes (solo se puede utilizar uno de ellos).

En el ejemplo anterior tendríamos una clase Punto que sería sólo visible (utilizable) desde el mismo paquete en el que se encuentra (modificador de acceso por omisión o de paquete, o package). Desde fuera de ese paquete no sería visible o accesible. Para usarla desde cualquier parte del código se añadiría el modificador "public".

## 2.3 Cuerpo de una clase

El cuerpo de una clase se encuentra encerrado entre llaves y contiene la declaración e implementación de sus miembros. Los miembros de una clase pueden ser:

- Atributos, que especifican los datos que podrá contener un objeto de la clase.
- Métodos, que implementan las acciones que se podrán realizar con un objeto.

Una clase puede no contener en su declaración atributos o métodos, pero debe de contener al menos uno de los dos (la clase no puede ser vacía). En el ejemplo anterior donde se definía una clase Punto, tendríamos los siguientes atributos:

- Atributo x, de tipo int.
- Atributo y, de tipo int.

Es decir, dos valores de tipo entero. Cualquier objeto de la clase Punto almacenará dos números enteros (x e y). Cada objeto distinto de la clase Punto contendrá sendos valores x e y, que podrán coincidir o no con los de otros objetos de la misma clase. Por ejemplo:

```
Punto p1, p2, p3;
```

Cada uno de esos objetos p1, p2 y p3 contendrá un par de coordenadas (x, y) que definen el estado de ese objeto. Puede que esos valores coincidan con los de otros objetos de tipo Punto o no, pero todos serán objetos creados a partir del mismo molde (clase). La clase Punto también define una serie de métodos:

```
int obtenerX () { return x; }  
int obtenerY() { return y; }  
void establecerX (int vx) { x= vx; };  
void establecerY (int vy) { y= vy; };
```

Cada método puede ser llamado desde cualquier objeto que sea una instancia de la clase Punto. Se trata de operaciones que permiten manipular los datos (atributos) contenidos en el objeto bien para calcular otros datos o bien para modificar los propios atributos.

## 2.4 Miembros estáticos o de clase

Cada vez que se produce una instancia de una clase (es decir, se crea un objeto de esa clase), se desencadenan los procesos (construcción del objeto) de creación en memoria de un espacio físico que, constituirá el objeto creado. De esta manera cada objeto tendrá sus propios miembros a imagen y semejanza de la plantilla propuesta por la clase.

En ocasiones, determinados miembros de la clase (atributos o métodos) no tienen mucho sentido como partes del objeto, sino más bien como partes de la clase en sí (partes de la plantilla, pero no de cada instancia de esa plantilla). Por ejemplo, la clase Coche podría disponer de un atributo con el nombre de la clase (de tipo String con la cadena "Coche"). No tendría sentido replicar el atributo para todos los objetos de la clase Coche, ya que siempre el mismo valor (la cadena "Coche"). Es más, podría tener sentido y existencia al margen de la existencia de cualquier objeto de tipo Coche. Podría no haberse creado ningún objeto de la clase Coche y seguiría teniendo sentido el atributo de nombre de la clase. Se trata de un atributo de la propia clase, no de cada objeto instancia de la clase.

Para poder definir miembros estáticos en Java se utiliza el modificador static. Los miembros (tanto atributos como métodos) declarados utilizando este modificador son conocidos como miembros estáticos o miembros de clase.

### 3 Atributos

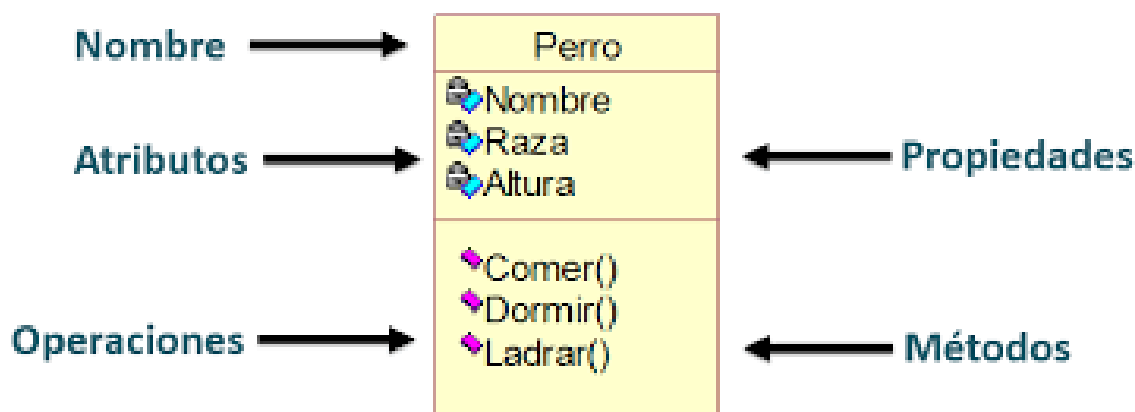
Los atributos constituyen la estructura interna de los objetos de una clase. Se trata del conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Es decir es como si fueran variables cuyo ámbito de existencia es el objeto dentro del cual han sido creadas. Fuera del objeto esas variables no tienen sentido y si el objeto deja de existir, esas variables también deberían hacerlo (proceso de destrucción del objeto). Los atributos a veces también son conocidos con el nombre de variables miembro o variables de objeto.

Los atributos pueden ser de cualquier tipo de los que pueda ser cualquier otra variable en un programa en Java: desde tipos elementales como int, boolean o float hasta tipos referenciados como arrays, Strings u objetos. Además del tipo y del nombre, la declaración de un atributo puede contener también algunos modificadores (como por ejemplo public, private, protected o static).

Por ejemplo, en el caso de la clase Punto que habíamos definido en el apartado anterior se podría haber declarado sus atributos como:

```
public int x;  
public int y;
```

De esta manera se estaría indicando que ambos atributos son públicos, es decir, accesibles por cualquier parte del código programa que tenga acceso a un objeto de esa clase. Lo habitual es declarar todos los atributos (o al menos la mayoría) como privados (private) de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase.



### 3.1 Declaración de atributos

La sintaxis general para la declaración de un atributo en el interior de una clase es:

```
[modificadores] <tipo><nombreAtributo>;
```

Ejemplos:

```
int x;  
public int elementoX, elementoY;  
private int x, y, z;  
static double descuentoGeneral;  
final bool casado;
```

La declaración de los atributos en una clase es exactamente igual a la declaración de cualquier variable y similar a como se hace en cualquier lenguaje de programación. Es decir mediante la indicación del tipo y a continuación el nombre del atributo, pudiéndose declarar varios atributos del mismo tipo mediante una lista de nombres de atributos separada por comas.

La declaración de un atributo (o variable miembro o variable de objeto) consiste en la declaración de una variable que únicamente existe en el interior del objeto y por tanto su vida comenzará cuando el objeto comience a existir (el objeto sea creado). Esto significa que cada vez que se cree un objeto se crearán tantas variables como atributos contenga ese objeto en su interior (definidas en la clase, que es la plantilla o “molde” del objeto). Todas esas variables estarán encapsuladas dentro del objeto y sólo tendrán sentido dentro de él.

En el ejemplo que estamos utilizando de objetos de tipo Punto (instancias de la clase Punto), cada vez que se cree un nuevo Punto p1, se crearán sendos atributos x, y de tipo int que estarán en el interior de ese punto p1. Si a continuación se crea un nuevo objeto Punto p2, se crearán otros dos nuevos atributos x, y de tipo int que estarán esta vez alojados en el interior de p2. Y así sucesivamente...

Dentro de la declaración de un atributo puedes encontrar tres partes:

- **Modificadores.** Son palabras reservadas que permiten modificar la utilización del atributo (indicar el control de acceso, si el atributo es constante, si se trata de un atributo de clase, etc.). Los iremos viendo uno a uno.
- **Tipo.** Indica el tipo del atributo. Puede tratarse de un tipo primitivo (int, char, bool, double...) o bien de uno referenciado (objeto, array, etc.).
- **Nombre.** Identificador único para el nombre del atributo. Por convenio se suelen utilizar las minúsculas. En caso de que se trate de un identificador que contenga varias palabras, a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas. Por ejemplo: primerValor, valor, sumaTotal, etc...

Como se observa, los atributos de una clase también pueden contener modificadores en



su declaración (como sucedía al declarar la propia clase). Estos modificadores permiten indicar cierto comportamiento de un atributo a la hora de utilizarlo.

Entre los modificadores de un atributo podemos distinguir:

- Modificadores de acceso. Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno.
- Modificadores de contenido. No son excluyentes. Pueden aparecer varios a la vez.
- Otros modificadores: **transient** y **volatile**. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable.

Esta es la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores e indicando cuáles son incompatibles unos con otros:

```
[private | protected | public] [static] [final] [transient] [volatile]  
<tipo><nombreAtributo>;
```

### 3.2 Modificadores de acceso

Los modificadores de acceso disponibles en Java para un atributo son:

- Modificador de acceso **por omisión** (o de paquete). Si no se indica ningún modificador de acceso en la declaración del atributo, se utilizará este tipo de acceso. Se permite acceder desde todas las clases que estén dentro del mismo paquete (package) que esta clase (la que contiene el atributo que se está declarando). No requiere ninguna palabra reservada.
- Modificador de acceso **public**. Indica que cualquier clase tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (public).
- Modificador de acceso **private**. Indica que solo se puede acceder al atributo desde dentro de la propia clase. El atributo estará “oculto” para cualquier otra zona de código fuera de la clase en la que está declarado. El contrario a public.
- Modificador de acceso **protected**. En este caso se permitirá acceder al atributo desde cualquier subclase (se verá más adelante al estudiar la herencia) de la clase en la que se encuentre declarado el atributo y desde las clases del mismo paquete.

Vemos un resumen de los niveles de acceso que permite cada modificador:

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)				
public				
private				
protected				

### **EJERCICIO RESUELTO.**

Imagina que quieres escribir una clase que represente un rectángulo en el plano. Para ello has pensado en los siguientes atributos:

- Atributos *x1*, *y1*, que representan la coordenadas del vértice inferior izquierdo del rectángulo. Ambos de tipo *double* (números reales).
- Atributos *x2*, *y2*, que representan las coordenadas del vértice superior derecho del rectángulo. También de tipo *double* (números reales).

Con estos dos puntos (*x1*, *y1*) y (*x2*, *y2*) se puede definir perfectamente la ubicación de un rectángulo en el plano.

Escribe una clase que contenga todos esos atributos teniendo en cuenta que queremos que sea una clase visible desde cualquier parte del programa y que sus atributos sean también accesibles desde cualquier parte del código.

### **Respuesta:**

Dado que se trata de una clase que podrá usarse desde cualquier parte del programa, utilizaremos el modificador de acceso *public* para la clase:

```
public class Rectangulo
```

Los cuatro atributos que necesitamos también han de ser visibles desde cualquier parte, así que también se utilizará el modificador de acceso *public* para los atributos:

```
public double x1, y1; // Vértice inferior izquierdo  
public double x2, y2; // Vértice superior derecho
```

De esta manera la clase completa quedaría:

```
public class Rectangulo {  
    public double x1, y1; // Vértice inferior izquierdo  
    public double x2, y2; // Vértice superior derecho  
}
```

## **3.3 Modificadores de contenido**

Los modificadores de contenido no son excluyentes (pueden aparecer varios para un mismo atributo). Son los siguientes:

- Modificador **static**. Hace que el atributo sea común a los objetos de una misma clase. Toda la clase compartirá el atributo con igual valor. Es un caso de miembro

estático o miembro de clase: atributo estático, atributo de clase o variable de clase.

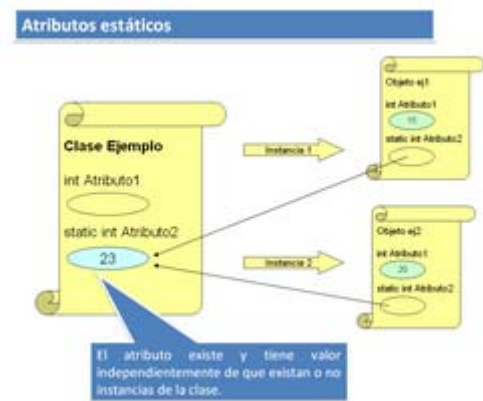
- Modificador **final**. Indica que el atributo es una constante. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los atributos constantes (final) se escribe con todas las letras en mayúsculas.

En el siguiente apartado sobre atributos estáticos se verá un ejemplo completo de un atributo estático (static). Veamos ahora un ejemplo de atributo constante (final).

```
class claseGeometria {  
    // Declaración de constantes  
    public final float PI= 3.14159265;  
    ...  
}
```

### 3.4 Atributos estáticos

Como ya hemos visto, el modificador static hace que el atributo sea común a todos los objetos de una misma clase. Podría decirse que la existencia del atributo no depende de la existencia del objeto, sino de la propia clase y por tanto sólo habrá uno, independientemente del número de objetos que se creen. El atributo será siempre igual para todos los objetos y tendrá un valor único, independientemente de cada objeto. Es más, aunque no exista ningún objeto de esa clase, el atributo sí existirá y podrá contener un valor (pues se trata de un atributo de la clase más que del objeto).



Uno de los ejemplos más habituales (y sencillos) de atributos estáticos o de clase es el de un contador que indica el número de objetos de esa clase que se han ido creando. Por ejemplo, en la clase Punto se podría incluir un atributo que fuera ese contador para llevar un registro del número de objetos de la clase Punto que se van construyendo durante la ejecución del programa.

Otro ejemplo de atributo estático (y en este caso también constante) que también se ha mencionado anteriormente al hablar de miembros estáticos era disponer de un atributo nombre, que contuviera un String con el nombre de la clase. Nuevamente ese atributo sólo tiene sentido para la clase, pues habrá de ser compartido por todos los objetos que sean de esa clase (es el nombre de la clase a la que pertenecen los objetos y por tanto siempre será la misma e igual para todos, no tiene sentido que cada objeto de tipo Punto almacene en su interior el nombre de la clase, eso lo debería hacer la propia clase).

```
class Punto {  
    // Coordenadas del punto  
    private int x, y;  
    // Atributos de clase: cantidad de puntos creados hasta el momento  
    public static cantidadPuntos;  
}
```

```
public static final nombre;
```

Obviamente, para que esto funcione, también habrá que escribir el código necesario para que cada vez que se cree un objeto de la clase Punto se incremente el valor del atributo cantidadPuntos.

### **EJERCICIO RESUELTO.**

*Ampliar el ejercicio anterior del rectángulo incluyendo los siguientes atributos:*

- *Atributo numRectangulos, que almacena el número de objetos de tipo rectángulo creados hasta el momento.*
- *Atributo nombre, que almacena el nombre de cada rectángulo.*
- *Atributo nombreFigura, que almacena el nombre de la clase, "Rectángulo".*
- *Atributo PI, que contiene el nombre de la constante PI con una precisión de cuatro cifras decimales.*

*No se desea que los atributos nombre y numRectangulos puedan ser visibles desde fuera de la clase. Y además se desea que la clase sea accesible solamente desde su propio paquete.*

### **Solución:**

*Los atributos numRectangulos, nombreFigura y PI podrían ser estáticos pues se trata de valores más asociados a la propia clase que a cada uno de los objetos que se puedan ir creando. Además, en el caso de PI y nombreFigura, también podría ser un atributo final, pues se trata de valores únicos y constantes (3.1416 en el caso de PI y "Rectángulo" en el caso de nombreFigura).*

*Dado que no se desea que se tenga accesibilidad a los atributos nombre y numRectangulos desde fuera de la clase podría utilizarse el atributo private para cada uno de ellos.*

*Por último hay que tener en cuenta que se desea que la clase sólo sea accesible desde el interior del paquete al que pertenece, por tanto habrá que utilizar el modificador por omisión o de paquete. Esto es, no incluir ningún modificador de acceso en la cabecera de la clase.*

*Teniendo en cuenta todo lo anterior la clase podría quedar finalmente así:*

```
class Rectangulo {  
    // Sin modificador "public" para que solo sea  
    // accesible desde el paquete  
    // Atributos de clase  
    private static int numRectangulos; // Número de rectángulos creados  
    public static final String nombreFigura= "Rectángulo"; // Nombre clase  
    public static final double PI= 3.1416; // Constante PI  
    // Atributos de objeto  
    private String nombre; // Nombre del rectángulo  
    public double x1, y1; // Vértice inferior izquierdo  
    public double x2, y2; // Vértice superior derecho  
}
```

## 4 Métodos

Como hemos visto, los métodos permiten definir el comportamiento de un objeto en sus interacciones con otros objetos. Forman parte de la estructura interna del objeto junto con los atributos. Al declarar una clase hemos visto cómo definirla y especificar sus atributos. Nos falta declarar los métodos, que estarán también en el interior del cuerpo de la clase junto con los atributos. Aunque atributos y métodos pueden aparecer mezclados en el interior del cuerpo de la clase, es aconsejable no hacerlo para mejorar la claridad y la legibilidad del código. De ese modo, al observar el código de una clase, veremos rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después.

Cada método puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar. Los métodos representan la interfaz de una clase (la manera en que otros objetos pueden comunicarse con un objeto determinado, solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción. Este modo de programar facilita mucho la tarea al desarrollador de aplicaciones, permitiéndole abstraerse del contenido de las clases, al hacer uso únicamente de su interfaz (métodos).

### 4.1 Declaración de un método

La definición de un método se compone de dos partes:

- Cabecera del método: contiene el nombre del método junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.
- Cuerpo del método: contiene las sentencias que implementan el comportamiento del método (incluidas posibles sentencias de declaración de variables locales).

Los elementos mínimos que deben aparecer en la declaración de un método son :

- El tipo devuelto por el método.
- El nombre del método.
- Los paréntesis.
- El cuerpo del método en tre llaves: { }

Por ejemplo, en la clase Punto que se ha estado utilizando en los apartados anteriores podemos encontrar el siguiente método:

```
int obtenerX ()  
{  
    // Cuerpo del método  
    ...  
}
```

Donde el tipo devuelto por el método es int, el nombre del método es "obtenerX" y no recibe ningún parámetro (aparece una lista vacía entre paréntesis. El cuerpo del método es todo el código encerrado entre llaves: { }

Dentro del cuerpo del método aparecen declaraciones de variables, sentencias y todo tipo de estructuras de control (bucles, condiciones, etc.) estudiadas en los apartados anteriores. La declaración de un método puede incluir algunos elementos más, que veremos más adelante.

## 4.2 Cabecera de un método

La declaración de un método puede incluir los siguientes elementos:

1. Modificadores (como los ya vistos, "public" o "private"). No es obligatorio incluir modificadores en la declaración.
2. El tipo devuelto (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si el tipo devuelto es "void", el método no devolverá ningún valor.
3. El nombre del método, aplicándose para los nombres el mismo convenio que para los atributos.
4. Una lista de parámetros separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros, la lista estará vacía y únicamente aparecerán los paréntesis.
5. Una lista de excepciones que el método puede lanzar. Se utiliza la palabra reservada "throws" seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente. Se verá en unidades posteriores.
6. El cuerpo del método, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

La sintaxis general de la cabecera de un método podría entonces quedar así:

```
[private|protected|public] [static] [abstract] [final] [native] [synchronized]  
<tipo><nombreMétodo> ( [<lista_parametros>] )  
[throws<lista_excepciones>]
```

Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: utilizar un **verbo en minúscula** o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos o nombres que comienzan con mayúsculas. Algunos ejemplos de métodos que siguen este convenio podrían ser: ejecutar, mover, subir, obtenerX, establecerValor, estaVacio, etc.

En el ejemplo de la clase Punto se puede ver que los métodos obtenerX y obtenerY siguen el convenio de nombres, devuelven en ambos casos un tipo int, su lista de parámetros es vacía (no tienen parámetros) y no lanzan ningún tipo de excepción:

```
int obtenerX ()  
int obtenerY ()
```

### 4.3 Modificadores en la declaración de un método

En la declaración de un método también pueden aparecer modificadores (como en la declaración de la clase o de los atributos). Un método puede tener los siguientes tipos de modificadores:

- Modificadores de acceso. Iguales que en el caso de los atributos: por omisión o de paquete, public, private y protected. Tienen el mismo cometido (definen si se puede acceder al método solo por parte de clases del mismo paquete, por cualquier parte del programa, etc).
- Modificadores de contenido. Como en el caso de los atributos, se usan los modificadores static y final, aunque tiene un significado diferente
- Otros modificadores (no son aplicables a los atributos, sólo a los métodos): abstract, native, synchronized.

Un **método static** es el que se implementa de la misma forma para todos los objetos de la clase. Solo tiene acceso a los atributos estáticos de la clase, dado que se trata de un método de clase y no de objeto. Solo podrá acceder a la información de clase y no a la de un objeto en particular. Este tipo de métodos pueden ser llamados sin necesidad de tener un objeto de la clase instanciado.

En Java, un ejemplo típico de métodos estáticos se encuentra en la clase Math, cuyos métodos son todos estáticos (Math.abs, Math.sin, Math.cos, etc.). La llamada a métodos estáticos se hace normalmente usando el nombre de la propia clase y no el de una instancia (objeto), pues se trata realmente de un **método de clase**. En cualquier caso, los objetos también admiten la invocación de los métodos estáticos de su clase

Un **método final** es un método que no permite ser sobrescrito por las clases descendientes de la clase a la que pertenece el método. Se verá en detalle en relación al concepto de herencia.

El modificador **native** es utilizado para señalar que un método ha sido implementado en código nativo (en un lenguaje compilado a lenguaje máquina, como por ejemplo C o C++). En estos casos simplemente se indica la cabecera del método, pues no tiene cuerpo escrito en Java.

Un **método abstract** no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un método abstract solo es posible dentro de una clase abstract. También se verá más adelante, con el concepto de herencia.



Por último, si un método ha sido declarado como **synchronized**, el entorno de ejecución obligará a que, cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método tengan que esperar a que el proceso termine. Puede resultar útil si sabemos que un determinado método va a poder ser llamado concurrentemente por varios procesos a la vez.

Veamos un resumen de todos modificadores vistos hasta ahora y su ámbito de aplicación:

	Clase	Atributo	Método
Sin modificador (paquete)			
public			
private			
protected			
static			
final			
synchronized			
native			
abstract			

#### 4.4 Parámetros en un método

La lista de parámetros de un método se coloca tras el nombre, entre paréntesis. Se escribe en parejas como "<tipoParametro><nombreParametro>". Cada uno de esos pares estará separado por comas:

```
<tipo> nombreMetodo ( <tipo_1><nombreParametro_1>, <tipo_2>  
<nombreParametro_2>, ..., <tipo_n><nombreParametro_n> )
```

Si la lista de parámetros es vacía, solo aparecerán los paréntesis:

```
<tipo><nombreMetodo> ( )
```

Al declarar un método, se debe tener en cuenta:

- Se puede incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía.
- Los parámetros podrán ser de cualquier tipo (tipos primitivos, objetos, arrays, etc.).
- No está permitido que el nombre de una variable local del método coincida con el nombre de un parámetro.
- No puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad.



- Si el nombre de algún parámetro coincide con el nombre de un atributo de la clase, éste será ocultado por el parámetro. Es decir, al indicar ese nombre en el código del método se hace referencia al parámetro, no al atributo. Para acceder al atributo hay que usar el operador de autorreferencia **this**, que veremos más adelante.
- En Java el paso de parámetros es siempre por valor, excepto en los tipos referenciados (por ejemplo, objetos) en cuyo caso se pasa una referencia. La referencia (el objeto en sí mismo) no podrá ser cambiada, pero sí sus elementos (atributos) a través de métodos o por acceso directo si es un miembro público.

Es posible utilizar una construcción especial llamada **varargs** (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

```
<tipo><nombreMetodo> (<tipo>... <nombre>)
```

También es posible mezclar el uso de varargs con parámetros fijos. En tal caso, la lista de parámetros variables debe aparecer al final (y solo puede aparecer una).

En realidad es una manera transparente de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que recorrer el array para obtener cada uno de los elementos de la lista de argumentos variables.

## 4.5 Cuerpo de un método

El interior de un método (cuerpo) está formado por sentencias en lenguaje Java:

- Sentencias de declaración de variables locales al método.
- Sentencias que implementan la lógica del método (estructuras de control como bucles o condiciones; utilización de métodos de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores.
- Sentencia de devolución del valor de retorno (return). Aparecerá al final del método y permite devolver la información que se le ha pedido al método. Es la última parte del proceso y la forma de comunicarse el código que llamó al método (paso de mensaje de vuelta). Se omite en el caso de que el tipo devuelto por el método sea void (vacío).

En la clase Punto teníamos los métodos **obtenerX** y **obtenerY**. Veamos uno de ellos:

```
int obtenerX ()  
{  
    return x;  
}
```

En ambos casos el método devuelve un valor (utilización de la sentencia return). No

recibe parámetros (mensajes o información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Solo devuelve el contenido de un atributo. Se trata de uno de los métodos más sencillos que se pueden implementar: un método que devuelve el valor de un atributo. Se les suele llamar métodos de tipo **get**.

Además de esos dos métodos, la clase también disponía de otros dos que sirven para la función opuesta (**establecerX** y **establecerY**):

```
void establecerX (int vx)
{
    x= vx;
}
```

Se trata de pasar un valor al método (parámetro vx de tipo int), que servirá para modificar el contenido del atributo x del objeto. Ahora no se devuelve ningún valor (el tipo devuelto es void, no hay sentencia return). Se llaman métodos de tipo **set**.

El método **establecerY** se usa del mismo modo. El código en el interior de un método suele ser algo más complejo, formado por un conjunto de sentencias en las que se realizan cálculos, se toman decisiones, se repiten acciones, etc.

```
class CCmporvalor
{
    public static void main(String args[])
    {
        int a=10;
        System.out.println(a);
        SumarNum(a);
        System.out.println(a);
    }

    static int SumarNum(int e)
    {
        e+=e;
        System.out.println(e);
        return e;           //Este método retorna un valor
    }
}
```

Veamos un ejemplo completo en un ejercicio:

### EJERCICIO RESUELTO.

Vamos a seguir ampliando la clase en la que se representa un rectángulo en el plano (clase Rectangulo). Para ello, pensemos en los siguientes métodos públicos:

- Métodos **obtenerNombre** y **establecerNombre**, que permiten el acceso y modificación del atributo nombre del rectángulo.
- Método **calcularSuperficie**, que calcula el área encerrada por el rectángulo.
- Método **calcularPerímetro**, que calcula el perímetro del rectángulo.
- Método **desplazar**, que mueve la ubicación del rectángulo en el plano en una cantidad X (para el ejeX) y otra cantidad Y (para el eje Y). Se trata simplemente de sumar el desplazamiento X a las coordenadas x1 y x2, y el desplazamiento Y a las coordenadas y1 e y2. Los parámetros de entrada de este método serán por tanto X e Y, de tipo double.
- Método **obtenerNumRectangulos**, que devuelve el número de rectángulos creados hasta el momento.

Incluye la implementación de cada uno de esos métodos en la clase Rectangulo.

Respuesta:

En el caso del método `obtenerNombre`, se trata de devolver el valor del atributo:

```
public String obtenerNombre () {  
    return nombre;  
}
```

El método `establecerNombre` también es muy sencillo:

```
public void establecerNombre (String nom) {  
    nombre= nom;  
}
```

Los métodos de cálculo de superficie y perímetro no van a recibir ningún parámetro de entrada, solo deben realizar cálculos a partir de los atributos contenidos en el objeto para obtener los resultados perseguidos. En cada caso habrá que aplicar la expresión matemática apropiada:

- En el caso de la superficie, habrá que calcular la longitud de la base y la altura del rectángulo a partir de las coordenadas de las esquinas inferior izquierda (x1, y1) y superior derecha (x2, y2) de la figura. La base sería la diferencia entre x2 y x1, y la altura la diferencia entre y2 e y1. A continuación se usa la fórmula "base x altura".
- En el caso del perímetro habrá también que calcular la longitud de la base y de la altura del rectángulo y a continuación sumar dos veces la longitud de la base y dos veces la longitud de la altura.

En ambos casos el resultado final tendrá que ser devuelto a través de la sentencia `return`. También es aconsejable en ambos casos el uso de variables locales para almacenar los cálculos intermedios (como `labase` o `la altura`).

```
public double calcularSuperficie () {  
double area, base, altura; // Variables locales  
// Cálculo de la base  
base= x2-x1;  
// Cálculo de la altura  
altura= y2-y1;  
// Cálculo del área  
area= base * altura;  
// Devolución del valor de retorno  
return area;  
}  
public double calcularPerimetro () {  
double perimetro, base, altura; // Variables locales  
// Cálculo de la base  
base= x2-x1;  
// Cálculo de la altura  
altura= y2-y1;  
// Cálculo del perímetro  
perimetro= 2*base + 2*altura;  
// Devolución del valor de retorno  
return perimetro;  
}
```

En el caso del método *desplazar*, se trata de modificar:

- Los contenidos de los atributos *x1* y *x2* sumándoles el parámetro *X*
- Los contenidos de los atributos *y1* e *y2* sumándoles el parámetro *Y*.

```
public void desplazar (double X, double Y) {  
// Desplazamiento en el eje X  
x1= x1 + X;  
x2= x2 + X;  
// Desplazamiento en el eje Y  
y1= y1 + Y;  
y2= y2 + Y;  
}
```

En este caso no se devuelve ningún valor (tipo devuelto vacío: *void*).

Por último, el método **obtenerNumRectangulos** simplemente debe devolver el valor del atributo **numRectangulos**. En este caso es razonable plantearse que este método podría ser más bien un método de clase(estático) más que un método de objeto, pues en realidad es una característica de la clase más que algún objeto en particular. Para ello tan solo tendrías que utilizar el modificador de acceso *static*:

```
public static int obtenerNumRectangulos () {  
return numRectangulos;  
}
```

## 4.6 Sobrecarga de métodos

Se puede pensar que un método aparece una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene por qué ser siempre así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga de métodos**.

Java soporta la característica conocida como sobrecarga de métodos, que permite declarar en una misma clase varias versiones de un mismo método con el mismo nombre. El compilador será capaz de distinguir entre ellos en función de la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Supongamos que desarrollamos una clase para escribir sobre un lienzo con diferentes tipografías según el tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (int), un número real (double) o una cadena de caracteres (String). Podemos definir un método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

```
Método pintarEntero (int entero).  
Método pintarReal (double real).  
Método pintarCadena (double String).  
Método pintarEnteroCadena (int entero, String cadena).
```

Y así sucesivamente para todos los casos que se puedan contemplar...

La sobrecarga ofrece la posibilidad de usar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: pintar). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo pintar, para todos los métodos anteriores:

- Método pintar (int entero)
- Método pintar (double real)
- Método pintar (double String)
- Método pintar (int entero, String cadena)

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes. Lo que sí habría producido un error de compilación habría sido, por ejemplo, incluir otro método pintar (int entero), pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método pintar con un único parámetro de tipo int).

También hemos de tener en cuenta que el tipo devuelto por el método no es considerado a la hora de identificar un método, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no se pueden definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no lo permitirá.

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), porque puede hacer el código menos legible.

## 4.7 Sobrecarga de operadores

Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como +, -, \*, ( ), <, >, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.

En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (+) o el producto (\*) para operar con fracciones. Si se definen objetos de una clase Fracción (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (+) para la suma, el operador asterisco (\*) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo Fracción mediante el algoritmo específico de suma o de producto del objeto Fracción (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

En algunos lenguajes de programación como por ejemplo C++ o C se permite la sobrecarga, pero no es algo soportado en todos los lenguajes.

El lenguaje Java no soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo Fracción, habrá que declarar métodos en la clase Fracción que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)
public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

## 4.8 La referencia this

La palabra reservada **this** consiste en una referencia al objeto actual. El uso de este operador puede ser muy útil para evitar la ambigüedad entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre).

En tales casos el parámetro “oculta” al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia **this** nos permite acceder a estos atributos ocultos por los parámetros.

Dado que **this** es una referencia a la propia clase en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto (.) como sucede con cualquier otra clase u objeto. Por tanto, en lugar de poner el nombre del atributo (que estos casos haría referencia al parámetro), se puede escribir `this.nombreAtributo`, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

En el ejemplo de la clase Punto, podríamos utilizar la referencia **this** si el nombre del parámetro del método coincidiera con el del atributo que se desea modificar. Por ejemplo:

```
void establecerX (int x)
{
    this.x= x;
}
```

En este caso ha sido indispensable el uso de **this**, pues si no sería imposible saber en qué casos te estás refiriendo al parámetro x y en cuáles al atributo x. Para el compilador el identificador x será siempre el parámetro, pues ha “ocultado” al atributo.

### **EJERCICIO RESUELTO.**

Modificar el método `obtenerNombre` de la clase `Rectangulo` de ejercicios anteriores utilizando la referencia **this**.

#### **Respuesta:**

Si utilizamos la referencia **this** en este método, entonces podremos utilizar como identificador del parámetro el mismo identificador que tiene el atributo (aunque no tiene porqué hacerse si no se desea):

```
public void establecerNombre (String nombre) {
    this.nombre= nombre;
}
```

## 4.9 Métodos estáticos

Como ya hemos visto en ocasiones anteriores, un método estático es un método que puede ser usado directamente desde la clase, sin necesidad de crear una instancia para poder usarlo. También se conoce como **método de clase**.

Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase, más que para una instancia concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:

- Acceso a atributos específicos de clase: incremento o decremento de contadores internos de la clase (no instancias), acceso a un posible atributo de nombre de la clase, etc.
- Operaciones genéricas relacionadas con la clase pero que no utilizan atributos de instancia. Por ejemplo una clase NIF (o DNI) que permite trabajar con el DNI y la letra del NIF y que proporciona funciones adicionales para calcular la letra NIF de un número de DNI que se le pase como parámetro. Ese método puede ser interesante para ser usado desde fuera de la clase de manera independiente a la existencia de objetos de tipo NIF.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete **java.lang** que representan tipos (Integer, String, Float, Double, Boolean, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

- `static String valueOf (int i)`. Devuelve la representación en formato String (cadena) de un valor int. Se trata de un método que no tiene que ver nada en absoluto con instancias concretas de String, sino de un método auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la clase. Por ejemplo:  
`String enteroCadena= String.valueOf(23).`
- `static String valueOf (float f)`. Similar para un valor de tipo float. Ejemplo:  
`StringfloatCadena= String.valueOf (24.341).`
- `static int parseInt (String s)`. Método estático de la clase Integer. Analiza la cadena pasada como parámetro y la transforma en un int. Ejemplo:  
`int cadenaEntero=Integer.parseInt ("-12").`

Todos los ejemplos anteriores son casos en los que se utiliza directamente la clase como una especie de caja de herramientas que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser métodos públicos.



## 5 Encapsulación

### 5.1 Ocultación de atributos. Métodos de acceso

Los atributos de una clase suelen ser declarados como privados a la clase o, como mucho, **protected** (accesibles también por clases heredadas), pero no como public. De esta manera se evita su manipulación inadecuada desde el exterior del objeto.

En estos casos se declaran los atributos como privados o protegidos y se crean métodos públicos que permiten acceder a esos atributos. Si se trata de un atributo cuyo contenido puede ser observado pero no modificado directamente, puede implementarse un método de “obtención” del atributo (método de tipo get) y si el atributo puede ser modificado, puede también implementar otro método para la modificación o “establecimiento” del valor del atributo (método de tipo set).

Si seguimos con la clase Punto usada como ejemplo:

```
private int x, y;  
// Métodos get  
public int obtenerX () { return  
public int obtenerY () { return  
// Métodos set  
public void establecerX (int x)  
public void establecerY (int y)  
x; }  
y; }  
{ this.x= x; }  
{ this.y= y; }
```

Así, para poder obtener el valor del atributo x de un objeto de tipo Punto será necesario utilizar el método *obtenerX()* y no se podrá acceder directamente al atributo x del objeto. Estos métodos se suelen llamar con expresiones como: getX ,getY(), setX, setY, getNombre, setNombre, getColor, etc.

También pueden darse casos en los que no interesa que pueda observarse directamente el valor de un atributo, sino un determinado procesamiento o cálculo que se haga con el atributo (pero no el valor original). Por ejemplo podrías tener un atributo DNI que almacene los 8 dígitos del DNI pero no la letra del NIF (pues se puede calcular a partir de los dígitos). El método de acceso para el DNI (método getDNI) podría proporcionar el DNicompleto (es decir, el NIF, incluyendo la letra), mientras que la letra no es almacenada realmente en el atributo del objeto. Algo similar podría suceder con el dígito de control de una cuenta bancaria, que puede no ser almacenado en el objeto, pero sí calculado y devuelto cuando se nos pide el número de cuenta completo.

En otros casos puede interesar disponer de métodos de modificación de un atributo pero a través de un determinado procesamiento previo para por ejemplo poder controlar errores o valores inadecuados. Volviendo al ejemplo del NIF, un método para modificar un

DNI (método setDNI) podría incluir la letra (NIF completo), de manera que así podría comprobarse si el número de DNI y la letra coinciden (es un NIF válido). En tal caso se almacenará el DNI y en caso contrario se producirá un error de validación (por ejemplo lanzando una excepción). En cualquier caso, el DNI que se almacenara sería solamente el número y no la letra (pues la letra es calculable a partir del número de DNI).

## 5.2 Ocultación de métodos

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

En el ejemplo anterior de objetos que contienen un DNI, será necesario calcular la letra correspondiente a un determinado número de DNI o comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos como métodos protegidos).

### EJERCICIO RESUELTO.

*Vamos a intentar implementar una clase que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un DNI español y que tenga las siguientes características:*

- *La clase almacenará el número de DNI en un int, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: private int numDNI.*
- *Para acceder al DNI se dispondrá de dos métodos obtener (get), uno que proporcionará el número deDNI (sólo las cifras numéricas) y otro que devolverá el NIF completo (incluida la letra). El formato del método será:*
  - *public int obtenerDNI ()*
  - *public String obtenerNIF ()*
- *Para modificar el DNI se dispondrá de dos métodos establecer (set), que permitirán modificar el DNI. Uno en el que habrá que proporcionar el NIF completo (número y letra). Y otro en el que únicamente será necesario proporcionar el DNI (las siete u ocho cifras). Si el DNI/NIF es incorrecto se debería lanzar algún tipo de excepción. El formato de los métodos (sobrecargados) será:*
  - *public void establecer (String nif) throws ...*
  - *public void establecer (int dni) throws ...*
- *La clase dispondrá de algunos métodos internos privados para calcular la letra de un número de DNI. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos*

(pertenecientes a la clase). Formato de los métodos:

- `private static char calcularLetraNIF (int dni)`
  - `private boolean validarNIF (String nif)`
  - `private static char extraerLetraNIF (String nif)`
  - `private static int extraerNumeroNIF (String nif)`
- Para calcular la letra NIF correspondiente a un número de DNI puedes consultar el artículo sobre el NIF de la Wikipedia: [http://es.wikipedia.org/wiki/N%C3%BAmero\\_de\\_identificaci%C3%B3n\\_fiscal](http://es.wikipedia.org/wiki/N%C3%BAmero_de_identificaci%C3%B3n_fiscal)

**Respuesta:**

```
/**-----  
 * Clase DNI  
-----*/  
public class DNI {  
    // Atributos estáticos  
    // Cadena con las letras posibles del DNI ordenados  
    private static final String LETRAS_DNI= "TRWAGMYFPDXBNJZSQVHLCKE";  
    // Atributos de objeto  
    private int numDNI;  
    // Métodos  
    public String obtenerNIF () {  
        // Variables locales  
        String cadenaNIF; // NIF con letra para devolver  
        char letraNIF; // Letra del número de NIF calculado  
        // Cálculo de la letra del NIF  
        letraNIF= calcularLetraNIF (numDNI);  
        // Construcción de la cadena del DNI: número + letra  
        cadenaNIF= Integer.toString(numDNI) + String.valueOf(letraNIF);  
        // Devolución del resultado  
        return cadenaNIF;  
    }  
    public int obtenerDNI () {  
        return numDNI;  
    }  
    public void establecer (String nif) throws Exception {  
        if (DNI.validarNIF (nif)) { // Valor válido: lo almacenamos  
            this.numDNI= DNI.extraerNumeroNIF(nif);  
        }  
        else { // Valor inválido: lanzamos una excepción  
            throw new Exception ("NIF inválido: " + nif);  
        }  
    }  
}
```

```
public void establecer (int dni) throws Exception {  
    // Comprobación de rangos
```

```
if (dni>99999999 && dni<999999999) {  
this.numDNI= dni; // Valor válido: lo almacenamos  
}  
else { // Valor inválido: lanzamos una excepción  
throw new Exception ("DNI inválido: " + String.valueOf(dni));  
}  
}  
private static char calcularLetraNIF (int dni) {  
char letra;  
letra= LETRAS_DNI.charAt(dni % 23); // Cálculo de la letra NIF  
return letra; // Devolución de la letra NIF  
}  
private static char extraerLetraNIF (String nif) {  
char letra=nif.charAt(nif.length()-1);  
return letra;  
}  
private static int extraerNumeroNIF (String nif) {  
int numero= Integer.parseInt(nif.substring(0, nif.length()-1));  
return numero;  
}  
private static boolean validarNIF (String nif) {  
boolean valido= true; // SuponemosNIF válido mientras no se vea fallo  
char letra_calculada;  
char letra_leida;  
int dni_leido;  
if (nif == null) { // El parámetro debe ser un objeto no vacío  
valido= false;  
}  
else if (nif.length()<8 || nif.length()>9) {  
valido= false;  
}  
else {  
letra_leida= DNI.extraerLetraNIF (nif); // Extraemos la letra de NIF  
dni_leido= DNI.extraerNumeroNIF (nif); // Extraemos el número de DNI  
letra_calculada= DNI.calcularLetraNIF(dni_leido); // Calculamos la letra  
if (letra_leida == letra_calculada) {  
// Comparamos la letra extraída con la calculada  
valido= true; // Supera las comprobaciones, el NIF es válido.  
}  
else {  
valido= false;  
}  
}  
return valido;  
}  
}
```

## 6 Utilización de métodos y atributos de una clase

### 6.1 Declaración de un objeto

Como ya hemos visto en unidades anteriores, la declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

```
<tipo> nombreVariable;
```

En este caso el tipo será alguna clase que ya hayas implementado o bien alguna de las proporcionadas por la biblioteca de Java o por alguna otra biblioteca escrita por terceros. Por ejemplo:

```
Punto p1;  
Rectangulo r1, r2;  
Coche cocheAntonio;  
String palabra;
```

Esas variables (p1, r1, r2, cocheAntonio, palabra) en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen “referencia”) a un objeto (una zona de memoria) de la clase indicada en la declaración. Un objeto recién declarado (referencia recién creada) no apunta a nada. Se dice que la referencia está vacía o que es una referencia nula (la variable objeto contiene el valor `null`). Es decir, la variable existe y está preparada para guardar una dirección de memoria que será la zona donde se encuentre el objeto al que hará referencia, pero el objeto aún no existe (no ha sido creado o instanciado). Por tanto se dice que apunta a un objeto nulo o inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto) es necesario crear o instanciar el objeto. Para ello se utiliza el operador `new`.

#### **EJERCICIO RESUELTO.**

*Utilizando la clase Rectangulo implementada en ejercicios anteriores, indica como declararías tres objetos (variables) de esa clase llamados r1, r2, r3.*

#### **Respuesta**

*Se trata simplemente de realizar una declaración de esas tres variables:*

*Rectangulo r1;*

*Rectangulo r2;*

*Rectangulo r3;*

*También se podría haber declarado los tres objetos en la misma sentencia de declaración:*

*Rectangulo r1, r2, r3;*

## 6.2 Creación de un objeto

Para poder crear un objeto (instancia de una clase) es necesario utilizar el operador new:

```
nombreObjeto= new <ConstructorClase> ([listaParametros]);
```

El constructor de una clase (ConstructorClase) es un método especial que tiene toda clase, cuyo nombre coincide con el de la clase. Construye el objeto, reservando la memoria necesaria para los atributos e inicializándolos si fuera necesario. El constructor podrá tener también su lista de parámetros. De la tarea de reservar memoria para la estructura del objeto (sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución. Es decir, al ejecutar un método constructor, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, enlace de la variable objeto a esa zona, etc.) y se pondrá rápidamente a desempeñarlas.

Cuando se escribe el código de una clase no es necesario implementar el método constructor. Java dota de un constructor por omisión (también conocido como constructor por defecto) a toda clase. Ese constructor por omisión se ocupará exclusivamente de las tareas de reserva de memoria. Si se desea que el constructor realice otras tareas adicionales, hay que escribirlo. El constructor por omisión no tiene parámetros.

El constructor por defecto no se ve en el código de una clase. Lo incluirá el compilador de Java al compilar la clase si descubre que no se ha creado ningún método constructor para esa clase. Algunos ejemplos de instanciación o creación de objetos podrían ser:

```
p1= new Punto ();  
r1= new Rectangulo ();  
r2= new Rectangulo;  
cocheAntonio= new Coche();  
palabra= String;
```

Si los constructores no tienen parámetros pueden omitirse los paréntesis vacíos. Un objeto puede ser declarado e instanciado en la misma línea. Por ejemplo:

```
Punto p1= new Punto ();
```

### **EJERCICIO RESUELTO.**

*Ampliar el ejercicio anterior instanciando los objetos r1, r2, r3 mediante el constructor por defecto.*

#### **Respuesta**

*Habría que añadir simplemente una sentencia de creación o instanciación (llamada al constructor mediante el operador new) por cada objeto que se desee crear:*

```
Rectangulo r1, r2,  
r1= new Rectangulo();  
r2= new Rectangulo();  
r3= new Rectangulo();
```

### 6.3 Manipulación de un objeto: utilización de métodos y atributos

Una vez que un objeto ha sido declarado y creado (clase instanciada) ya sí se puede decir que el objeto existe en el entorno de ejecución, y por tanto puede ser manipulado como un objeto más en el programa, haciéndose uso de sus atributos y sus métodos. Para acceder a un miembro de un objeto se utiliza el operador punto (.) del siguiente modo:

`<nombreObjeto>.<nombreMiembro>`

Donde `<nombreMiembro>` será el nombre de algún miembro del objeto (atributo o método) al cual se tenga acceso.

Por ejemplo, en el caso de los objetos de tipo Punto que has declarado e instanciado en los apartados anteriores, podrías acceder a sus miembros de la siguiente manera:

```
Punto p1, p2, p3;  
p1= new Punto();  
p1.x= 5;  
p1.y= 6;  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.x, p1.y);  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());  
p1.establecerX(25);  
p1.establecerX(30);  
System.out.printf ("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
```

Es decir, colocando el operador punto (.) a continuación del nombre del objeto y seguido del nombre del miembro al que se desea acceder.

#### **EJERCICIO RESUELTO.**

Utilizar el ejemplo de los rectángulos para crear un rectángulo *r1*, asignarle los valores *x1=0*, *y1=0*, *x2=10*, *y2=10*, calcular su área y su perímetro y mostrarlos en pantalla.

#### **Respuesta**

Se trata de declarar e instanciar el objeto *r1*, rellenar sus atributos de ubicación (coordenadas de las esquinas), e invocar a los métodos *calcularSuperficie* y *calcularPerimetro* utilizando el operador punto (.). Por ejemplo:

```
Rectangulo r1= new Rectangulo ();  
r1.x= 0;  
r1.y= 0;  
r2.x= 10;  
r2.y= 10;  
area= r1.calcularSuperficie ();  
perímetro= r1.calcularPerimetro ();
```

Por último faltaría mostrar en pantalla la información calculada.

Podemos observar a continuación un ejemplo completo donde se instancia un objeto *Rectangulo* y se manipulan sus miembros:

*Clase principal o main...*

```
/*
Ejemplo de uso de la clase Rectangulo
*/
package ejemplorectangulos01;
/**
 *
 * Programa Principal (clase principal)
 */
public class EjemploRectangulos01 {
public static void main(String[] args) {
Rectangulo r1, r2;
r1= new Rectangulo ();
r2= new Rectangulo ();
r1.x1= 0;
r1.y1= 0;
r1.x2= 10;
r1.y2= 10;
r1.establecerNombre ("rectangulo1");
System.out.printf ("PRUEBA DE USO DE LA CLASE RECTÁNGULO\n");
System.out.printf ("-----\n\n");
System.out.printf ("r1.x1: %4.2f\nr1.y1: %4.2f\n", r1.x1, r1.y1);
System.out.printf ("r1.x2: %4.2f\nr1.y2: %4.2f\n", r1.x2, r1.y2);
System.out.printf
("Perimetro:%4.2f\nSuperficie:%4.2f\n",r1.CalcularPerimetro(),
r1.CalcularSuperficie());
System.out.printf ("Desplazamos X=3, Y=3\n");
r1.desplazar (3,3);
System.out.printf ("r1.x1: %4.2f\nr1.y1: %4.2f\n", r1.x1, r1.y1);
System.out.printf ("r1.x2: %4.2f\nr1.y2: %4.2f\n", r1.x2, r1.y2);
}
}
```

*Clase rectángulo...*

```
package ejemplorectangulos01;
/**-----
 * Clase Rectangulo
-----*/
public class Rectangulo {
// Atributos de clase
private static int numRectangulos;
public static final String nombreFigura= "Rectángulo";
public static final double PI= 3.1416;
// Atributos de objeto
private String nombre;
public double x1, y1;
public double x2, y2;
```



```
// Método obtenerNombre
public String obtenerNombre () {
return nombre;
}
// Método establecerNombre
public void establecerNombre (String nom) {
nombre= nom;
}
// Método CalcularSuperficie
public double CalcularSuperficie () {
double area, base, altura;
// Cálculo de la base
base= x2-x1;
// Cálculo de la altura
altura= y2-y1;
// Cálculo del área
area= base * altura;
// Devolución del valor de retorno
return area;
}
// Método CalcularPerimetro
public double CalcularPerimetro () {
double perimetro, base, altura;
// Cálculo de la base
base= x2-x1;
// Cálculo de la altura
altura= y2-y1;
// Cálculo del perímetro
perimetro= 2*base + 2*altura;
// Devolución del valor de retorno
return perimetro;
}
// Método desplazar
public void desplazar (double X, double Y) {
// Desplazamiento en el eje X
x1= x1 + X;
x2= x2 + X;
// Desplazamiento en el eje Y
y1= y1 + Y;
y2= y2 + Y;
}
// Método obtenerNumRectangulos
public static int obtenerNumRectangulos () {
return numRectangulos;
}
}
```

## 6.4 Utilización de constructores

Un ejemplo de constructor para la clase Punto podría ser:

```
public Punto (int x, int y)
{
    this.x= x;
    this.y= y;
    cantidadPuntos++; // Suponiendo que tengamos un atributo estático
    cantidadPuntos
}
```

En este caso el constructor recibe dos parámetros. Además de reservar espacio para los atributos (de lo cual se encarga automáticamente Java), también asigna sendos valores iniciales a los atributos x e y. Por último incrementa un atributo (probablemente estático) llamado *cantidadPuntos*.

Una vez que dispongas de tus propios constructores personalizados, la forma de utilizarlos es igual que con el constructor por defecto (mediante la utilización de la palabra reservada new) pero teniendo en cuenta que si has declarado parámetros en tu método constructor, tendrás que llamar al constructor con algún valor para esos parámetros.

Un ejemplo de utilización del constructor para la clase Punto del apartado anterior sería:

```
Punto p1;
p1= new Punto (10, 7);
```

En este caso no se estaría utilizando el constructor por defecto sino el constructor que acabamos de implementar en el cual además de reservar memoria se asigna un valor a algunos de los atributos.

### **EJERCICIO RESUELTO.**

*Ampliar el ejercicio de la clase Rectangulo añadiéndole tres constructores:*

1. *Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1);*
2. *Un constructor con cuatro parámetros, x1, y1, x2, y2, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.*
3. *Un constructor con dos parámetros, base y altura, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.*

### Respuesta

En el caso del primer constructor lo único que hay que hacer es "rellenar" los atributos x1,y1,x2,y2 con los valores 0, 0, 1, 1:

```
public Rectangulo() {  
    x1=0.0;  
    y1=0.0;  
    x2=1.0;  
    y2=1.0;  
}
```

Para el segundo constructor es suficiente con asignar a los atributos x1, y1, x2, y2 los valores de los parámetros x1, y1, x2, y2. Tan solo hay que tener en cuenta que al tener los mismos nombres los parámetros del método que los atributos de la clase, estos últimos son ocultados por los primeros y para poder tener acceso a ellos habrá que utilizar el operador de autorreferencia this:

```
public Rectangulo (double x1, double y1, double x2, double y2)  
{  
    this.x1= x1;  
    this.y1= y1;  
    this.x2= x2;  
    this.y2= y2;  
}
```

En el caso del tercer constructor habrá que inicializar el vértice (x1,y1) a (0,0) y el vértice (x2,y2) a (0 + base, 0 + altura), es decir a (base, altura):

```
public Rectangulo (double base, double altura)  
{  
    this.x1= 0.0;  
    this.y1= 0.0;  
    this.x2= base;  
    this.y2= altura;  
}
```

Veamos ahora el ejemplo completo:

```
/*  
 * Ejemplo de uso de la clase Rectangulo con constructores  
 */  
package ejemplorectangulos02;  
/**  
 *  
 * Programa Principal (clase principal)  
 */  
public class EjemploRectangulos02 {  
    public static void main(String[] args) {  
        Rectangulo r1, r2, r3;  
        System.out.printf ("PRUEBA DE USO DE LA CLASE RECTÁNGULO\n");
```

```
System.out.printf ("-----\n\n");
System.out.printf ("Creando rectángulos...\n\n");
r1= new Rectangulo ();
r2= new Rectangulo (1,1, 3,3);
r3= new Rectangulo (10, 5);
System.out.printf ("Rectángulo r1: \n");
System.out.printf ("r1.x1: %4.2f\nr1.y1: %4.2f\n",r1.x1, r1.y1);
System.out.printf ("r1.x2: %4.2f\nr1.y2: %4.2f\n",r1.x2, r1.y2);
System.out.printf ("Perímetro: %4.2f\nSuperficie: %4.2f\n",
r1.CalcularPerimetro(),r1.CalcularSuperficie());
System.out.printf ("Rectángulo r2: \n");
System.out.printf ("r2.x1: %4.2f\nr2.y1: %4.2f\nr2.y1: %4.2f\n",r2.x1,r2.y1);
System.out.printf ("r2.x2: %4.2f\nr2.y2: %4.2f\nr2.y2: %4.2f\n",r2.x2,r2.y2);
System.out.printf("Perímetro: %4.2f\nSuperficie: %4.2f\n",
r2.CalcularPerimetro(),r2.CalcularSuperficie());
System.out.printf ("Rectángulo r3: \n");
System.out.printf ("r3.x1: %4.2f\nr3.y1: %4.2f\n",r3.x1, r3.y1);
System.out.printf ("r3.x2: %4.2f\nr3.y2: %4.2f\n",r3.x2, r3.y2);
System.out.printf ("Perímetro: %4.2f\nSuperficie: %4.2f\n",
r3.CalcularSuperficie());
}
}
```

Clase Rectangulo

package ejemplorectangulos02;

```
/**-----
 * Clase Rectangulo.
 * Incluye constructores.
 *-----*/

public class Rectangulo {
// Atributos de clase (estáticos)
private static int numRectangulos; // Número total de rectángulos
public static final String nombreFigura= "Rectángulo";
public static final double PI= 3.1416;
// Atributos de objeto
private String nombre; // Nombre del rectángulo
public double x1, y1; // Vértice inferior izquierdo
public double x2, y2; // Vértice superior derecho
```

```
//-----
```

```
// Constructores
//-----
public Rectangulo ()
{
    x1=0.0;
    y1=0.0;
    x2=1.0;
    y2=1.0;
}
public Rectangulo (double x1, double y1, double x2, double y2)
{
    this.x1= x1;
    this.y1= y1;
    this.x2= x2;
    this.y2= y2;
}
public Rectangulo (double base, double altura)
{
    this.x1= 0.0;
    this.y1= 0.0;
    this.x2= base;
    this.y2= altura;
}
//-----
// Métodos estáticos (de clase)
//-----
// Métodos de estáticos públicos
// -----
// Método obtenerNumRectangulos
public static int obtenerNumRectangulos () {
    return numRectangulos;
}
//-----
// Métodos de objeto
//-----
//Métodos públicos
//-----
// Método obtenerNombre
public String obtenerNombre () {
    return nombre;
}
// Método establecerNombre
public void establecerNombre (String nom) {
    nombre= nom;
}
```

```
// Método CalcularSuperficie
public double CalcularSuperficie () {
double area, base, altura;
// Cálculo de la base
base= x2-x1;
// Cálculo de la altura
altura= y2-y1;
// Cálculo del área
area= base * altura;
// Devolución del valor de retorno
return area;
}
// Método CalcularPerimetro
public double CalcularPerimetro () {
double perimetro, base, altura;
// Cálculo de la base
base= x2-x1;
// Cálculo de la altura
altura= y2-y1;
// Cálculo del perímetro
perimetro= 2*base + 2*altura;
// Devolución del valor de retorno
return perimetro;
}
// Método desplazar
public void desplazar (double X, double Y) {
// Desplazamiento en el eje X
x1= x1 + X;
x2= x2 + X;
// Desplazamiento en el eje Y
y1= y1 + Y;
y2= y2 + Y;
}
}
```

## 6.5 Constructores de copia

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase

que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras clonar el objeto tantas veces como haga falta. A este mecanismo se le llama constructor copia o constructor de copia.

Un **constructor copia** es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y copia todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

Un ejemplo de constructor copia para la clase Punto podría ser:

```
public Punto (Punto p)
{
    this.x= p.obtenerX();
    this.y= p.obtenerY();
}
```

En este caso el constructor recibe como parámetro un objeto del mismo tipo que el que va a ser creado (clasePunto), inspecciona el valor de sus atributos (atributos x e y), y los reproduce en los atributos del objeto en proceso de construcción (this).

Un ejemplo de utilización de ese constructor podría ser:

```
Punto p1, p2;
p1= new Punto (10, 7);
p2= new Punto (p1);
```

En este caso el objeto p2 se crea a partir de los valores del objeto p1.

#### **EJERCICIO RESUELTO.**

*Ampliar el ejercicio de la clase Rectangulo añadiéndole un constructor copia.*

#### **Respuesta**

*Se trata de añadir un nuevo constructor además de los tres que ya habíamos creado:*

```
// Constructor copia
public Rectangulo (Rectangulo r) {
    this.x1= r.x1;
    this.y1= r.y1;
    this.x2= r.x2;
    this.y2= r.y2;
}
```

*Para usar este constructor basta con haber creado anteriormente otro Rectangulo para utilizarlo como base de la copia. Por ejemplo:*

```
Rectangulo r1, r2;
r1= new Rectangulo (0,0,2,2);
r2= new Rectangulo (r1);
```

## 7 Introducción a la herencia

La herencia es uno de los conceptos fundamentales que introduce la programación orientada a objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características (reutilización).

A una clase que hereda de otra se le llama subclase o clase hija y aquella de la que se hereda es conocida como superclase o clase padre. También se puede hablar en general de clases descendientes o clases ascendientes. Al heredar, la subclase adquiere todas las características (atributos y métodos) de su superclase, aunque algunas de ellas pueden ser sobrescritas o modificadas dentro de la subclase (a eso se le suele llamar especialización).

Una clase puede heredar de otra que a su vez ha podido heredar de una tercera y así sucesivamente. Esto significa que las clases van tomando todas las características de sus clases ascendientes (no sólo de su superclase o clase padre inmediata) a lo largo de toda la rama del árbol de la jerarquía de clases en la que se encuentre.

Imagina que quieres modelar el funcionamiento de algunos vehículos para trabajar con ellos en un programa de simulación. Lo primero que haces es pensar en una clase Vehículo que tendrá un conjunto de atributos (por ejemplo: posición actual, velocidad actual y velocidad máxima que puede alcanzar el vehículo) y de métodos (por ejemplo: detener, acelerar, frenar, establecerDirección, establecer sentido).

Dado que vas a trabajar con muchos tipos de vehículos, no tendrás suficiente con esas características, así que seguramente vas a necesitar nuevas clases que las incorporen. Pero las características básicas que has definido en la clase Vehículo van a ser compartidas por cualquier nuevo vehículo que vayas a modelar. Esto significa que si creas otra clase podrías heredar de Vehículo todas esos atributos y propiedades y tan solo tendrías que añadir las nuevas.

Si vas a trabajar con vehículos que se desplazan por tierra, agua y aire, tendrás que idear nuevas clases con características adicionales. Por ejemplo, podrías crear una clase VehiculoTerrestre, que herede las características de Vehículo, pero que también incorpore atributos como el número de ruedas o la altura de los bajos). A su vez, podría idearse una nueva clase que herede de VehiculoTerrestre y que incorpore nuevos atributos y métodos como, por ejemplo, una clase Coche.

Y así sucesivamente con toda la jerarquía de clases heredadas que consideres oportunas para representar lo mejor posible el entorno y la información sobre la que van a trabajar tus programas.



## Utilización de clases heredadas

¿Cómo se indica en Java que una clase hereda de otra? Para indicar que una clase hereda de otra es necesario utilizar la palabra reservada `extends` junto con el nombre de la clase de la que se quieren heredar sus características:

```
class <NombreClase> extends <nombreSuperClase> {  
...  
}
```

En el ejemplo anterior de los vehículos, la clase `VehiculoTerrestre` podría quedar así al ser declarada:

```
class VehiculoTerrestre extends Vehículo {  
...  
}
```

Y en el caso de la clase `Coche`:

```
class Coche extends VehiculoTerrestre {  
...  
}
```

En unidades posteriores estudiarás detalladamente cómo crear una jerarquía de clases y qué relación existe entre la herencia y los distintos modificadores de clases, atributos y métodos. Por ahora es suficiente con que entiendas el concepto de herencia y sepas reconocer cuándo una clase hereda de otra (uso de la palabra reservada `extends`).

En las bibliotecas proporcionadas por Java aparecen jerarquías bastante complejas de clases heredadas en las cuales se han ido aprovechando cada uno de los miembros de una clase base para ir construyendo las distintas clases derivadas añadiendo (y a veces modificando) poco a poco nueva funcionalidad.

Eso suele suceder en cualquier proyecto de software conforme se van a analizando, descomponiendo y modelando los datos con los que hay que trabajar. La idea es poder representar de una manera eficiente toda la información que es manipulada por el sistema que se desea automatizar. Una jerarquía de clases suele ser una buena forma de hacerlo.

En el caso de Java, cualquier clase con la que trabajes tendrá un ascendiente. Si en la declaración de clase no indicas la clase de la que se hereda (no se incluye un `extends`), el compilador considerará automáticamente que se hereda de la clase `Object`, que es la clase que se encuentra en el nivel superior de toda la jerarquía de clases en Java (y que es la única que no hereda de nadie).

También irás viendo al estudiar distintos componentes de las bibliotecas de Java (por ejemplo en el caso de las interfaces gráficas) que para poder crear objetos basados en las clases proporcionadas por esas bibliotecas tendrás que crear tus propias clases que hereden de algunas de esas clases. Para ellos tendrás que hacer uso de la palabra reservada `extends`.

## 8 Empaquetado de clases

La encapsulación de la información dentro de las clases ha permitido llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Es posible que conforme vaya aumentando la complejidad de tus aplicaciones necesites que algunas de tus clases puedan tener acceso a parte de la implementación de otras debido a las relaciones que se establezcan entre ellas a la hora de diseñar tu modelo de datos. En estos casos se puede hablar de un nivel superior de encapsulamiento y ocultación conocido como empaquetado.

Un paquete consiste en un conjunto de clases relacionadas entre sí y agrupadas bajo un mismo nombre. Normalmente se encuentran en un mismo paquete todas aquellas clases que forman una biblioteca o que reúnen algún tipo de característica en común. Esto la organización de las clases para luego localizar fácilmente aquellas que vayas necesitando.

### 8.1 Jerarquía de paquetes

Los paquetes en Java pueden organizarse jerárquicamente de manera similar a lo que puedes encontrar en la estructura de carpetas en un dispositivo de almacenamiento, donde:

- Las clases serían como los archivos.
- Cada paquete sería como una carpeta que contiene archivos (clases).
- Cada paquete puede además contener otros paquetes (como las carpetas que contienen carpetas).
- Para poder hacer referencia a una clase dentro de una estructura de paquetes, habrá que indicar la trayectoria completa desde el paquete raíz de la jerarquía hasta el paquete en el que se encuentra la clase, indicando por último el nombre de la clase (como el path absoluto de un archivo).

La estructura de paquetes en Java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica. Por otro lado, la organización en paquetes permite también el control de acceso a miembros de las clases desde otras clases que estén en el mismo paquete gracias a los modificadores de acceso (recuerda que uno de los modificadores que viste era precisamente el de paquete).

Las clases que forman parte de la jerarquía de clases de Java se encuentran organizadas en diversos paquetes. Todas las clases proporcionadas por Java en sus bibliotecas son miembros de distintos paquetes y se encuentran organizadas jerárquicamente. Dentro de cada paquete hay un conjunto de clases con relaciones entre ellas. Ese conjunto de paquetes forman la API de Java. Las clases básicas del lenguaje se encuentran en el paquete `java.lang`, las clases de entrada/salida en `java.io` y en el paquete `java.math` se encuentran clases para trabajar con números grandes y de gran precisión.

## 8.2 Utilización de los paquetes

Es posible acceder a cualquier clase de cualquier paquete (siempre que ese paquete esté disponible en nuestro sistema, obviamente) mediante la calificación completa de la clase dentro de la estructura jerárquica de paquete. Es decir indicando la trayectoria completa de paquetes desde el paquete raíz hasta la propia clase. Eso se puede hacer utilizando el operador punto (.) para especificar cada subpaquete:

```
paquete_raiz.subpaquete1.subpaquete2. ... .subpaquete_n.NombreClase
```

Por ejemplo: `java.lang.String`.

En este caso se está haciendo referencia a la clase `String` que se encuentra dentro del paquete `java.lang`. Este paquete contiene las clases elementales para poder desarrollar una aplicación Java. Otro ejemplo podría ser: `java.util.regex.Patern`. En este otro caso se hace referencia a la clase `Patern` ubicada en el paquete `java.util.regex`, que contiene clases para trabajar con expresiones regulares.

Dado que puede resultar bastante tedioso tener que escribir la trayectoria completa de una clase cada vez que se quiera utilizar, existe la posibilidad de indicar que se desea trabajar con las clases de uno o varios paquetes. De esa manera cuando se vaya a utilizar una clase que pertenezca a uno de esos paquetes no será necesario indicar toda su trayectoria. Para ello se utiliza la sentencia `import` (importar):

```
import paquete_raiz.subpaquete1.subpaquete2...subpaquete_n.NombreClase;
```

Los ejemplos anteriores quedarían entonces:

```
import java.lang.String;
import java.util.regex.Patern;
```

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un `import` de cada una, podemos utilizar el comodín (símbolo asterisco: `"*"`) para indicar que queremos importar todas las clases de ese paquete y no sólo una determinada:

```
import java.lang.*;
import java.util.regex.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, sino solo las que haya en el paquete. La importación de las clases contenidas en los subpaquetes se indica explícitamente. Por ejemplo:

```
import java.util.*;
import java.util.regex.*;
```

En este caso se importarán todas las clases del paquete `java.util` (clases `Date`, `Calendar`, `Timer`, etc.) y de su subpaquete `java.util.regex` (`Matcher` y `Pattern`), pero las de otros subpaquetes como `java.util.concurrent` o `java.util.jar`.

En el caso del paquete `java.lang`, no es necesario realizar importación. El compilador, dada la importancia de este paquete, permite el uso de sus clases sin indicar la

trayectoria, como si se incluyera en la primera línea la sentencia `import java.lang.*`.

### 8.3 Inclusión de una clase en un paquete

Al principio de cada archivo .java se puede indicar a qué paquete pertenece mediante la palabra reservada `package` seguida del nombre del paquete:

```
package nombre_paquete;  
Por ejemplo:  
package paqueteEjemplo;  
class claseEjemplo {  
...  
}
```

La sentencia `package` debe ser incluida en cada archivo fuente de cada clase que quieras incluir ese paquete. Si en un archivo fuente hay definidas más de una clase, todas esas clases formarán parte del paquete indicado en la sentencia `package`.

Si al comienzo de un archivo Java no se incluyen ninguna sentencia `package`, el compilador considerará que las clases de ese archivo formarán parte del paquete por omisión (un paquete sin nombre asociado al proyecto).

Para evitar la ambigüedad, dentro de un mismo paquete no puede haber dos clases con el mismo nombre, aunque sí pueden existir clases con el mismo nombre si están en paquetes diferentes. El compilador será capaz de distinguir una clase de otra gracias a que pertenecen a paquetes distintos.

Como ya has visto en unidades anteriores, el nombre de un archivo fuente en Java se construye utilizando el nombre de la clase pública que contiene junto con la extensión .java, pudiendo haber únicamente una clase pública por cada archivo fuente. El nombre de la clase debía coincidir (en mayúsculas y minúsculas) exactamente con el nombre del archivo en el que se encontraba definida.

Así, si por ejemplo tenías una clase `Punto` dentro de un archivo `Punto.java`, la compilación daría lugar a un archivo `Punto.class`.

En el caso de los paquetes, la correspondencia es a nivel de directorios o carpetas. Es decir, si la clase `Punto` se encuentra dentro del paquete `prog.figuras`, el archivo `Punto.java` debería encontrarse en la carpeta `prog\figuras`. Para que esto funcione correctamente el compilador ha de ser capaz de localizar todos los paquetes (tanto los estándar de Java como los definidos por otros programadores).

Es decir, que el compilador debe tener conocimiento de dónde comienza la estructura de carpetas definida por los paquetes y en la cual se encuentran las clases. Para ello se utiliza el `ClassPath` cuyo funcionamiento habrás estudiado en las primeras unidades de este módulo. Se trata de una variable de entorno que contiene todas las rutas en las que comienzan las estructuras de directorios (distintas jerarquías posibles de paquetes) en las

que están contenidas las clases.

## 8.4 Proceso de creación de un paquete

Para crear un paquete en Java se siguen estos pasos:

1. Poner un nombre al paquete. Suele ser habitual utilizar el dominio de Internet de la empresa que ha creado el paquete. Por ejemplo, para el caso de `miempresa.com`, podría utilizarse un nombre de paquete `com.miempresa`.
2. Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes. La ruta de la raíz de esa estructura jerárquica deberá estar especificada en el ClassPath de Java.
3. Especificar a qué paquete pertenecen la clase (o clases) de el archivo `.java` mediante el uso de la sentencia `package`.

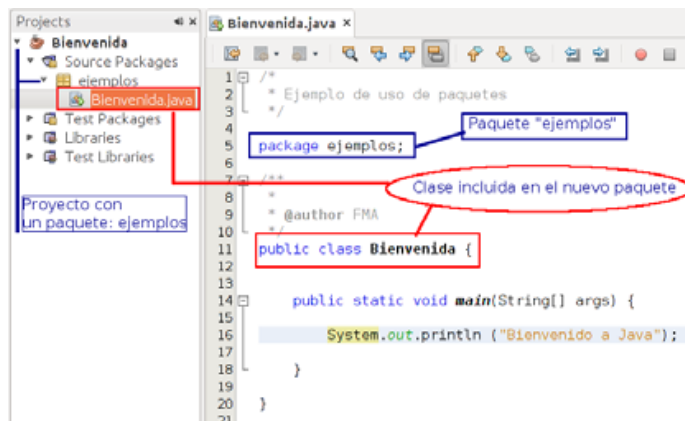
## 9 Librerías de objetos

A medida que los programas se hacen más grandes, el número de clases va creciendo. Es recomendable hacer grupos de clases, de forma que todas las clases que estén relacionadas o traten sobre un mismo tema estén en el mismo grupo.

Un paquete de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común. Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Por ello se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

Java nos ayuda a organizar las clases en paquetes. En cada fichero `.java` podemos indicar a qué paquete pertenece la clase que hagamos en ese fichero. Los paquetes se declaran utilizando la palabra clave `package` seguida del nombre del paquete. Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase: `package Nombre_de_Paquete;`

Por ejemplo, si decidimos agrupar en un paquete "ejemplos" un programa llamado "Bienvenida", pondríamos en nuestro fichero `Bienvenida.java` lo siguiente:



## 9.1 Sentencia import.

Cuando queremos utilizar una clase de un paquete distinto a la clase que estamos utilizando, se puede usar la sentencia import. Por ejemplo, para usar la clase Scanner del paquete java.util de la Biblioteca de Clases de Java, tendremos que escribir:

```
import java.util.Scanner;
```

Se pueden importar todas las clases de un paquete:

```
import java.awt.*;
```

Esta sentencia debe aparecer al principio de la clase, justo tras la sentencia package, si existe. También podemos utilizar la clase sin sentencia import:

```
java.util.Scanner teclado = new java.util.Scanner (System.in);
```

Al trabajar con paquetes, Java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma para que todo funcione adecuadamente.

## 9.2 Compilar y ejecutar clases con paquetes.

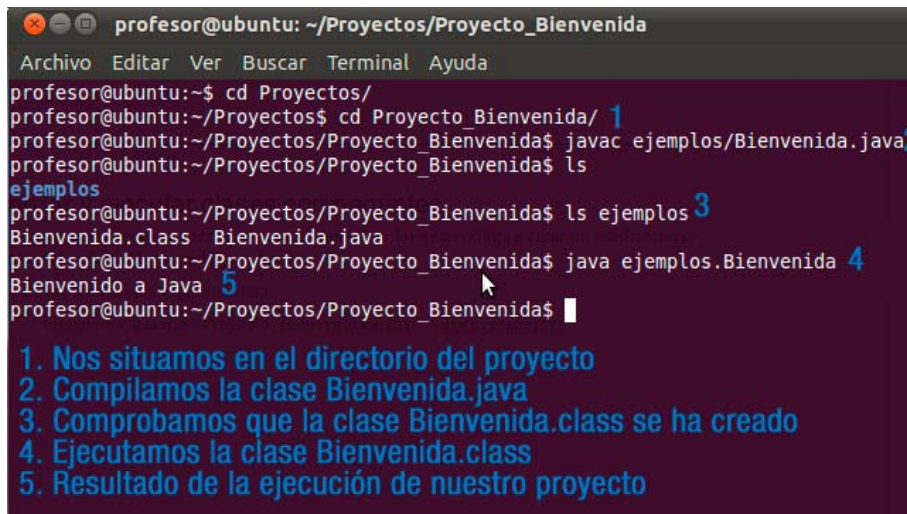
Si hacemos que Bienvenida.java pertenezca al paquete ejemplos, debemos crear un subdirectorio "ejemplos" y meter dentro el archivo Bienvenida.java. Por ejemplo, en Linux tendríamos esta estructura de directorios:

```
/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/Bienvenida.java
```

Debemos tener cuidado con las mayúsculas y las minúsculas. Para compilar la clase Bienvenida.java que está en el paquete ejemplos debemos situarnos en el directorio padre del paquete y compilar desde ahí:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida
$ javac ejemplos/Bienvenida.java
```

En el directorio ejemplos nos aparecerá la clase compilada Bienvenida.class.



```
profesor@ubuntu: ~/Proyectos/Proyecto_Bienvenida
Archivo Editar Ver Buscar Terminal Ayuda
profesor@ubuntu:~$ cd Proyectos/
profesor@ubuntu:~/Proyectos$ cd Proyecto_Bienvenida/ 1
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ javac ejemplos/Bienvenida.java 2
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ ls
ejemplos
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ ls ejemplos 3
Bienvenida.class Bienvenida.java
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$ java ejemplos.Bienvenida 4
Bienvenido a Java 5
profesor@ubuntu:~/Proyectos/Proyecto_Bienvenida$
```

1. Nos situamos en el directorio del proyecto  
2. Compilamos la clase Bienvenida.java  
3. Comprobamos que la clase Bienvenida.class se ha creado  
4. Ejecutamos la clase Bienvenida.class  
5. Resultado de la ejecución de nuestro proyecto

Para ejecutar la clase compilada Bienvenida.class que está en el directorio ejemplos, debemos seguir situados en el directorio padre del paquete. El nombre completo de la clase es "paquete.clase", es decir "ejemplos.Bienvenida". Los pasos serían los siguientes:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida
$ java ejemplos/Bienvenida
```

Si todo es correcto, debe salir el mensaje "Bienvenido a Java" por la pantalla.

### 9.3 Jerarquía de paquetes.

Para organizar mejor las cosas, un paquete, en vez de clases, también puede contener otros paquetes. Es decir, podemos hacer subpaquetes de los paquetes y así sucesivamente. Esto permite agrupar paquetes relacionados en un paquete más grande. Por ejemplo, si quiero dividir mis clases de ejemplos en ejemplos básicos y ejemplos avanzados, puedo poner más niveles de paquetes separando por puntos:

```
package ejemplos.basicos;
package ejemplos.avanzados;
```

A nivel de sistema operativo, tendríamos que crear los subdirectorios básicos y avanzados dentro del directorio ejemplos, y meter ahí las clases que correspondan. Para compilar, en el directorio del proyecto habría que compilar poniendo todo el path hasta llegar a la clase. Es decir, el nombre de la clase va con todos los paquetes separados por puntos, por ejemplo `ejemplos.basicos.Bienvenida`.

La estructura de directorios en el sistema operativo cuando usamos subpaquetes sería:  
`/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/basicos/HolaMundo.java`



Y la compilación y ejecución sería:

```
$ cd /<directorio_usuario>/Proyecto_Bienvenida
$ javac ejemplos/basicos/Bienvenida.java
$ java ejemplos/basicos/Bienvenida
Hola Mundo
```

La Biblioteca de Clases de Java se organiza haciendo uso de esta jerarquía de paquetes. Así por ejemplo, si quiero acceder a la clase `Date`, tendré que importarla indicando su ruta completa, o sea, `java.util.Date`, así:

```
import java.util.Date;
```

## 9.4 Librerías de Java.

La API de Java es un conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas. Utilizar las clases y métodos de la Biblioteca de Java nos va ayudar a reducir el tiempo de desarrollo considerablemente, por lo que es importante que aprendamos a consultarla y conozcamos las clases más utilizadas.

Los paquetes más importantes que ofrece el lenguaje Java son:

- **java.io.** Contiene las clases que gestionan la entrada y salida, ya sea para manipular ficheros, leer o escribir en pantalla, en memoria, etc. Este paquete contiene por ejemplo la clase `BufferedReader` que se utiliza para la entrada por teclado.
- **java.lang.** Contiene las clases básicas del lenguaje. Este paquete no es necesario importarlo, ya que es importado automáticamente por el entorno de ejecución. En este paquete se encuentra la clase `Object`, que sirve como raíz para la jerarquía de clases de Java, o la clase `System` que ya hemos utilizado en algunos ejemplos y que representa al sistema en el que se está ejecutando la aplicación. También podemos encontrar en este paquete las clases que "envuelven" los tipos primitivos de datos. Lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo las conversiones de datos.
- **java.util.** Biblioteca de clases de utilidad general para el programador. Este paquete contiene por ejemplo la clase `Scanner` utilizada para la entrada por teclado de diferentes tipos de datos, la clase `Date`, para el tratamiento de fechas, etc.
- **java.math.** Contiene herramientas para manipulaciones matemáticas.
- **java.awt.** Incluye las clases relacionadas con la construcción de interfaces de usuario, es decir, las que nos permiten construir ventanas, cajas de texto, botones, etc. Algunas de las clases que podemos encontrar en este paquete son `Button`, `TextField`, `Frame`, `Label`, etc.



- **java.swing.** Contiene otro conjunto de clases para la construcción de interfaces avanzadas de usuario. Los componentes que se engloban dentro de este paquete se denominan componentes Swing, y suponen una alternativa mucho más potente que AWT para construir interfaces de usuario.
- **java.net.** Conjunto de clases para la programación en la red local e Internet.
- **java.sql.** Contiene las clases necesarias para programar en Java el acceso a las bases de datos.
- **java.security.** Biblioteca de clases para implementar mecanismos de seguridad.

Como se puede comprobar Java ofrece una completa jerarquía de clases organizadas a través de paquetes.

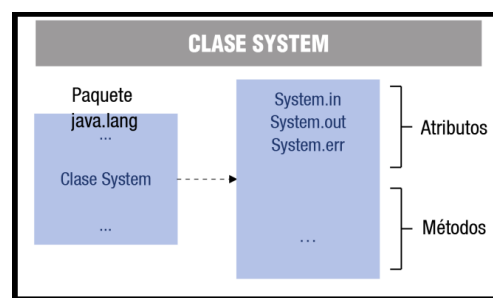
En el siguiente enlace se puede acceder a la información oficial sobre la Biblioteca de Clases de Java:

<http://www.oracle.com/technetwork/java/javase/documentation/api-jsp-136079.html>

## 10 Programación de consola: entrada y salida de información.

Los programas a veces necesitan acceder a los recursos del sistema, como por ejemplo los dispositivos de entrada/salida estándar, para recoger datos de teclado o mostrar datos por pantalla. En Java, la entrada por teclado y la salida de información por pantalla se hace mediante la clase **System** del paquete `java.lang` de la Biblioteca de Clases de Java. Como cualquier otra clase, está compuesta de métodos y atributos. Los atributos de la clase System son tres objetos que se utilizan para la entrada y salida estándar. Estos objetos son los siguientes:

- **System.in.** Entrada estándar: teclado.
- **System.out.** Salida estándar: pantalla.
- **System.err.** Salida de error estándar, se produce también por pantalla, pero se implementa como un fichero distinto al anterior para distinguir la salida normal del programa de los mensajes de error. Se utiliza para mostrar mensajes de error.



No se pueden crear objetos a partir de la clase System, sino que se utiliza directamente llamando a cualquiera de sus métodos con el operador de manipulación de objetos, es decir, el operador punto (.):

```
System.out.println("Bienvenido a Java");
```

En el siguiente enlace se pueden consultar los atributos y métodos de la clase System del paquete `java.lang` perteneciente a la Biblioteca de Clases de Java (versión 7):

<http://download.oracle.com/javase/7/docs/api/>

## 10.1 Conceptos sobre la clase System.

La lectura por teclado es muy importante cuando empezamos a hacer nuestros primeros programas. Para entender mejor en qué consiste la clase System, y en particular el objeto `System.in` vamos a describirlo más detenidamente.

En el apartado anterior hemos dicho que `System.in` es un atributo de la clase System, que está dentro del paquete `java.lang`. Pero además, si consultamos la Biblioteca de Clases de Java, nos damos cuenta que es un objeto, y como todos los objetos debe ser instanciado. En efecto, volviendo a consultar la biblioteca de clases nos damos cuenta que `System.in` es una instancia de una clase de java que se llama `InputStream`.

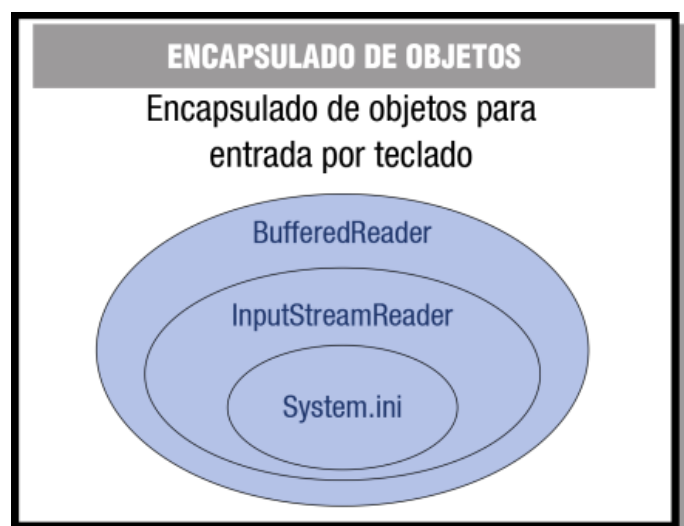
Field Summary	
Fields	
Modifier and Type	Field and Description
static <code>PrintStream</code>	<code>err</code> The "standard" error output stream.
static <code>InputStream</code>	<code>in</code> The "standard" input stream.
static <code>PrintStream</code>	<code>out</code> The "standard" output stream.

En Java, `InputStream` nos permite leer en bytes, desde teclado, un archivo o cualquier otro dispositivo de entrada. Con esta clase podemos utilizar por ejemplo el método `read()` que permite leer un byte de la entrada o `skip (long n)`, que salta n bytes de la entrada.

Pero lo que realmente nos interesa es poder leer texto o números, no bytes, para hacernos más cómoda la entrada de datos.

Para ello se utilizan las clases:

- **InputStreamReader.** Convierte los bytes leídos en caracteres. Particularmente, nos va a servir para convertir el objeto `System.in` en otro tipo de objeto que nos permita leer caracteres.
- **BufferedReader.** Lee hasta un fin de línea. Esta es la clase que nos interesa utilizar, pues tiene un método `readLine()` que nos va a permitir leer caracteres hasta el final de línea.



La forma de instanciar estas clases para usarlas con `System.in` es la siguiente:

```
InputStreamReader isr = new InputStreamReader(System.in);  
BufferedReader br = new BufferedReader (isr);
```

En el código anterior hemos creado un `InputStreamReader` a partir de `System.in` y pasamos dicho `InputStreamReader` al constructor de `BufferedReader`.

El resultado es que las lecturas que hagamos con el objeto `br` son en realidad realizadas sobre `System.in`, pero con la ventaja de que podemos leer una línea completa. Así, por ejemplo, si escribimos una A, con:

```
String cadena = br.readLine();
```

Obtendremos en cadena una "A". Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático `parseInt()` de la clase `Integer`, con lo cual la lectura la haríamos así:

```
int numero = Integer.parseInt (br.readLine());
```

## 10.2 Entrada por teclado. Clase System.

A continuación vamos a ver un ejemplo de cómo utilizar la clase `System` para la entrada de datos por teclado en Java. Para compilar y ejecutar el ejemplo existen las órdenes `javac` y `java`, aunque también se puede utilizar un IDE como Eclipse. El código sería:

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
/*  
 * Ejemplo de entrada por teclado con la clase System  
 */  
/**  
 *  
 * @author JJ  
 */  
public class EntradaTecladoSystem {  
    public static void main(String[] args) {  
        try{  
            InputStreamReader isr = new InputStreamReader(System.in);  
            BufferedReader br = new BufferedReader(isr);  
            System.out.print("Introduce el texto: ");  
            String cad = br.readLine();  
            //salida por pantalla del texto introducido  
            System.out.println(cad);  
            System.out.print("Introduce un numero: ");  
            int num = Integer.parseInt(br.readLine());  
            // salida por pantalla del numero introducido  
            System.out.println(num);  
        }  
    }  
}
```

```
} catch (Exception e) {  
// System.out.println("Error al leer datos");  
e.printStackTrace();  
}  
}  
}
```

Hemos metido el código entre excepciones try-catch. Cuando en nuestro programa falla algo, por ejemplo la conversión de un String a int, Java nos avisa lanzando excepciones. Si "capturamos" esa excepción en nuestro programa, podemos avisar al usuario de qué ha pasado. Esto es conveniente porque si no tratamos la System excepción seguramente el programa se pare y no siga ejecutándose.

El control de excepciones se verá en unidades posteriores, pero por ahora nos interesa saber que en las llaves del try colocamos el código que puede fallar y en las llaves del catch el tratamiento de la excepción.

### 10.3 Entrada por teclado. Clase Scanner.

Al manejar la entrada por teclado como en el apartado anterior, solo podemos leer fácilmente datos de tipo String. Para leer otros tipos de datos hay que usar conversiones de tipos. El kit de Desarrollo de Java, a partir de su versión 1.5, incorpora la clase java.util.Scanner, que permite leer tipos de datos String, int, long, etc., a través de la consola de la aplicación. Por ejemplo, para leer un tipo de datos entero por teclado sería:

```
Scanner teclado = new Scanner (System.in);  
int i = teclado.nextInt ();
```

O bien esta otra instrucción para leer una línea completa, incluido texto o números:

```
String cadena = teclado.nextLine();
```

Hemos creado un objeto de la clase Scanner, llamado teclado, utilizando el constructor de la clase, tomando como parámetro la entrada básica del sistema System.in, por defecto asociada al teclado. Veamos cómo funciona un objeto de la clase Scanner:

```
import java.util.Scanner;  
/*  
 * Ejemplo de entrada de teclado con la clase Scanner  
 *  
 * @author JJ  
 */  
public class EntradaTecladoScanner {  
public static void main(String[] args) {  
// Creamos objeto teclado  
Scanner teclado = new Scanner(System.in);  
// Declaramos variables a utilizar  
String nombre;
```

```
int edad;
boolean estudias;
float salario;
// Entrada de datos
System.out.println("Nombre: ");
nombre=teclado.nextLine();
System.out.println("Edad: ");
edad=teclado.nextInt();
System.out.println("Estudias: ");
estudias=teclado.nextBoolean();
System.out.println("Salario: ");
salario=teclado.nextFloat();
// Salida de datos
System.out.println("Bienvenido: " + nombre);
System.out.println("Tienes: " + edad + " años");
System.out.println("Estudias: " + estudias);
System.out.println("Tu salario es: " + salario + " euros");
}
```

## 10.4 Salida por pantalla.

La salida por pantalla en Java se hace con el objeto `System.out`. Este objeto es una instancia de la clase `PrintStream` del paquete `java.lang`. Si miramos la API de `PrintStream` obtendremos la variedad de métodos para mostrar datos por pantalla. Algunos son:

- `void print(String s)`: Escribe una cadena de texto.
- `void println(String x)`: Escribe una cadena de texto y termina la línea.
- `void printf(String format, Object... args)`: Escribe una cadena de texto utilizando formato.

En las órdenes `print` y `println`, cuando queramos escribir el valor de una variable debemos utilizar el operador de concatenación de cadenas (+), por ejemplo:

```
System.out.println("Bienvenido, " + nombre);
```

Este programa escribe el mensaje de "Bienvenido, Carlos", si el valor de la variable `nombre` es Carlos.

Las órdenes `print` y `println` consideran todas las variables como cadenas de texto sin formato. Por ejemplo, no se podría escribir un número decimal con dos cifras decimales o redondear las cifras, o escribir los puntos de los miles.

Para formatear los datos en la salida se utiliza la orden `printf()`. Esta orden usa códigos de conversión para indicar cómo mostrar el contenido. Estos códigos se caracterizan porque llevan delante el símbolo `%`. Algunos de estos códigos son:

`%c`: Escribe un carácter.

`%s`: Escribe una cadena de texto.

`%d`: Escribe un entero.

`%f`: Escribe un número en punto flotante.

`%e`: Escribe un número en punto flotante en notación científica.

Por ejemplo, si queremos escribir el número float 12345.1684 con el punto de los miles y sólo dos cifras decimales la orden sería:

```
System.out.printf("% ,.2f\n", 12345.1684);
```

Esta orden mostraría el número 12.345,17 por pantalla.

Estas órdenes pueden utilizar las secuencias de escape que vimos en unidades anteriores, como `"\n"` para crear un salto de línea, `"\t"` para introducir un salto de tabulación en el texto, etc.

## 10.5 Salida de error.

La salida de error está representada por el objeto `System.err`. Este objeto es también una instancia de la clase `PrintStream`, por lo que podemos utilizar los mismos métodos vistos anteriormente.

No parece muy útil utilizar `out` y `err` si su destino es la misma pantalla, o al menos en el caso de la consola del sistema donde las dos salidas son representadas con el mismo color y no notamos diferencia alguna. En cambio en la consola de varios entornos integrados de desarrollo como NetBeans o Eclipse la salida de `err` se ve en un color diferente.

## 11 Documentación del código

A lo largo de nuestra vida como programadores es probable que nos veamos en la necesidad de reutilizar, modificar y mantener nuestro propio código o incluso, código de otros desarrolladores. La generación de una documentación adecuada de nuestros programas puede suponer una inestimable ayuda para realizar ciertos procesos en el software.

Si analizamos la documentación de las clases proporcionada en los paquetes que distribuye Oracle, nos daremos cuenta de que dicha documentación ha sido generada con una herramienta llamada **Javadoc**. Pues bien, nosotros también podremos generar la documentación de nuestro código a través de dicha herramienta.

Si desde el principio nos acostumbramos a documentar el funcionamiento de nuestras clases desde el propio código fuente, estaremos facilitando la generación de la futura

documentación de nuestras aplicaciones. ¿Cómo lo logramos? A través de una serie de comentarios especiales, llamados **comentarios de documentación** que serán tomados por Javadoc para generar una serie de archivos HTML que permitirán posteriormente, navegar por nuestra documentación con cualquier navegador web.

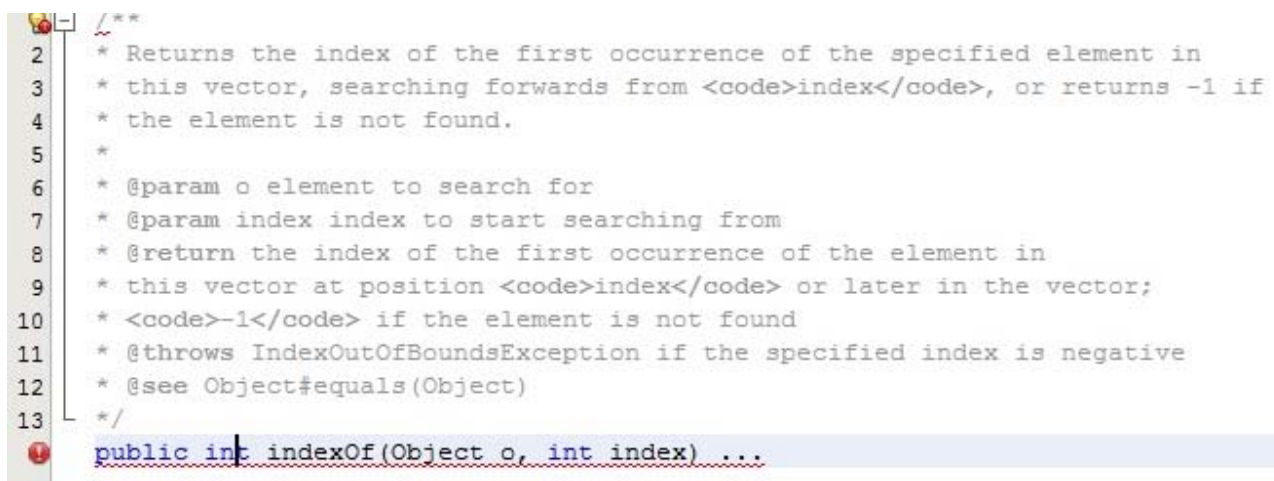
Los comentarios de documentación tienen una marca de comienzo (/\*\*) y una marca de fin (\*). En su interior podremos encontrar dos partes diferenciadas: una para realizar una descripción y otra en la que encontraremos más etiquetas de documentación. Veamos un ejemplo:

```
/**  
 * Descripción principal (texto/HTML)  
 *  
 * Etiquetas (texto/HTML)  
 */
```

Este es el formato general de un comentario de documentación. Comenzamos con la marca de comienzo en una línea. Cada línea de comentario comenzará con un asterisco. El final del comentario de documentación deberá incorporar la marca de fin. Las dos partes diferenciadas de este comentario son:

- **Zona de descripción:** es aquella en la que el programador escribe un comentario sobre la clase, atributo, constructor o método que se vaya a codificar bajo el comentario. Se puede incluir la cantidad de texto que se necesite, pudiendo añadir etiquetas HTML que formateen el texto escrito y así ofrecer una visualización mejorada al generar la documentación mediante Javadoc.
- **Zona de etiquetas:** en esta parte se colocará un conjunto de etiquetas de documentación a las que se asocian textos. Cada etiqueta tendrá un significado especial y aparecerán en lugares determinados de la documentación, una vez haya sido generada.

En la siguiente imagen puedes observar un ejemplo de un comentario de documentación.



```
1 /**  
2  * Returns the index of the first occurrence of the specified element in  
3  * this vector, searching forwards from <code>index</code>, or returns -1 if  
4  * the element is not found.  
5  *  
6  * @param o element to search for  
7  * @param index index to start searching from  
8  * @return the index of the first occurrence of the element in  
9  * this vector at position <code>index</code> or later in the vector;  
10 * <code>-1</code> if the element is not found  
11 * @throws IndexOutOfBoundsException if the specified index is negative  
12 * @see Object#equals(Object)  
13 */  
14 public int indexOf(Object o, int index) ...
```



## 11.1 Etiquetas y posición

Cuando hemos de incorporar determinadas etiquetas a nuestros comentarios de documentación es necesario conocer dónde y qué etiquetas colocar, según el tipo de código que estemos documentando en ese momento. Existirán dos tipos generales de etiquetas:

1. **Etiquetas de bloque:** Son etiquetas que sólo pueden ser incluidas en el bloque de documentación, después de la descripción principal y comienzan con el símbolo @.
2. **Etiquetas en texto:** Son etiquetas que pueden ponerse en cualquier punto de la descripción o en cualquier punto de la documentación asociada a una etiqueta de bloque. Son etiquetas definidas entre llaves, de la siguiente forma {@etiqueta}

En la siguiente tabla podrás encontrar una referencia sobre las diferentes etiquetas y su ámbito de uso.

	@autor	{@code}	{@docRoot}	@deprecated	@exception	{@@inheritDoc}	{@link}	{@literal}
Descripción	✓		✓	✓			✓	✓
Paquete	✓		✓	✓			✓	✓
Clases e Interfaces	✓		✓	✓			✓	✓
Atributos			✓	✓			✓	✓
Constructores y métodos			✓	✓	✓	✓	✓	

## 11.2 Uso de las etiquetas

Veamos la función de las etiquetas más habituales a la hora de generar comentarios de documentación:

- **@autor** texto con el nombre: Esta etiqueta sólo se admite en clases e interfaces. El texto después de la etiqueta no necesitará un formato especial. Podremos incluir tantas etiquetas de este tipo como necesitemos.
- **@version** texto de la versión: El texto de la versión no necesitará un formato especial. Es conveniente incluir el número de la versión y la fecha de ésta. Podremos incluir varias etiquetas de este tipo una detrás de otra.



- `@deprecated` texto: Indica que no debería utilizarse, indicando en el texto las causas de ello. Se puede utilizar en todos los apartados de la documentación. Si se ha realizado una sustitución debería indicarse qué utilizar en su lugar. Por ejemplo:

```
@deprecated El método no funciona correctamente. Se recomienda el uso de  
{@link metodoCorrecto}
```

- `@exception` nombre-excepción texto: Esta etiqueta es equivalente a `@throws`.
- `@param` nombre-atributo texto: Esta etiqueta es aplicable a parámetros de constructores y métodos. Describe los parámetros del constructor o método. Nombre-atributo es idéntico al nombre del parámetro. Cada etiqueta `@param` irá seguida del nombre del parámetro y después de una descripción de éste. Por ejemplo:

```
@param fromIndex El índice del 1er elemento que debe ser eliminado.
```

- `@return` texto: Esta etiqueta se puede omitir en los métodos que devuelven void. Deberá aparecer en todos los métodos, dejando explícito qué tipo o clase de valor devuelve y sus posibles rangos de valores. Veamos un ejemplo:

```
/**  
 * Chequea si un vector no contiene elementos.  
 *  
 * @return <code>verdadero</code>si solo si este vector no contiene  
 * componentes, esto es, su tamaño es cero;  
 * <code>falso</code> en cualquier otro caso.  
 */  
public boolean VectorVacio() {  
    return elementCount == 0;  
}
```

- `@see` referencia: Se aplica a clases, interfaces, constructores, métodos, atributos y paquetes. Añade enlaces de referencia a otras partes de la documentación. Podremos añadir a la etiqueta: cadenas de caracteres, enlaces HTML a páginas y a otras zonas del código. Por ejemplo:

```
@see "Diseño de patrones: La reusabilidad de los elementos de la  
programación orientada a objetos"
```

```
@see <a href="http://www.w3.org/WAI/">Reference Web</a>
```

```
@see String#equals(Object) equals
```

- `@throws` nombre-excepción texto: En nombre-excepción tendremos que indicar el nombre completo de ésta. Podremos añadir una etiqueta por cada excepción que se lance explícitamente con una cláusula `throws`, pero siguiendo el orden alfabético. Esta etiqueta es aplicable a constructores y métodos, describiendo las posibles excepciones del constructor/método.

## 11.3 Orden de las etiquetas

Las etiquetas deben disponerse en un orden determinado, que es el siguiente:

etiqueta	Descripción
@autor	En clases e interfaces. Se pueden poner varios. Es mejor ponerlas en orden cronológico.
@version	En clases e interfaces.
@param	En métodos y constructores. Se colocarán tantos como parámetros tenga el constructor o método. Mejor en el mismo orden en el que se encuentren declarados.
@return	En métodos.
@exception	En constructores y métodos. Mejor en el mismo orden en el que se han declarado, o en orden alfabético.
@throws	Es equivalente a @exception.
@see	Podemos poner varios. Comenzaremos por los más generales y después los más específicos.
@deprecated	

Para conocer cómo obtener a través de Javadoc la documentación de tus aplicaciones, observa el siguiente código:

```
/*
 * ejemplojavadoc.java
 *
 * Created on July 5, 2013, 11:24 AM
 */
/**
 * Clase que comienza la estructura de excepciones
 * @author JJ
 * @since 1.0
 * @see Visitar www.adictosaltrabajo.com
 */
class RCEException extends Exception
{
    void depura(String psError)
    {
        System.out.println("Error: " + psError);
    }

    RCEException(String psError)
    {
        super(psError);
        depura(psError);
    }
}
/**
```

```
*
* @author JJ
* @since 1.0
* @see Visitar www.madrid.org
*/
public class ejemplojavadoc {

    /** Constantes publicas
    public static final int
    public static final int
    public static final int
    de gestion errores*/

    ERROR = 0;
    LOG = 1;
    INFO = 2;

    /** Constructor por defecto */
    public ejemplojavadoc() {
    }

    void depura(String sError)
    {
        System.out.println("ejemplojavadoc: " + sError);
    }

    /**
    * @param args Array de argumentos
    */
    public static void main(String[] args) {
        /** Construimos un objeto no estático */
        ejemplojavadoc objetoAuxiliar = new ejemplojavadoc();
        try
        {
            objetoAuxiliar.ejecuta();
        }
        catch(RCException e)
        {
            objetoAuxiliar.depura("Excepcion = " + e.getMessage());
        }
    }

    /**
    * Punto de entrada a la aplicación
    * @exception RCException Se genera una excepción genérica.
    * @return true
    */
    public boolean ejecuta() throws RCException
```

```
{  
/** Retornamos true por defecto */  
int error = 0;  
if(error == 0)  
{  
throw new RuntimeException("Invocamos excepciones");  
}  
return true;  
}  
}
```