4. TEMA 4: Programación Orientada a Objetos. Clases y objetos.

Nos adentramos en la orientación a objetos. Una vez entendidas las bases de la programación, que hasta ahora lo visto está mas relacionado con la programación estructurada, estamos preparados para comprender los elementos, las bases y el funcionamiento de la programación orientada a objetos.

Conoceremos la estructura interna de una clase, la definición de sus atributos y métodos. Aprenderemos a instanciar clases y crear objetos.

Así mismo también los conceptos y bondades que nos proporciona la orientación a objetos, la encapsulación, reutilización, etc.

Se van a definir métodos estáticos. Visibilidad y alcance de variables globales y locales.

Una vez acabado este tema seremos capaces de dominar las bases de la orientación a objetos y crear programas mas complejos.

4.1 ¿Porqué la programación orientada a objetos?

La realización de un programa es la respuesta a un problema. Para resolver un problema tenemos que descomponerlo en casos más sencillos y en tareas individuales para luego volver a construir, a unir, todas estas partes dándole solución al problema.

El criterio de descomposición del sistema ha sido el funcional identificando las funciones del sistemas y sus partes teniendo así partes más simples. Este método da buen resultado cuando dichas funciones están bien definidas y son estables en el tiempo. Si un sistema recibe cambios estructurales grandes esta descomposición en funciones se tambalea teniendo que, seguramente, hacer cambios profundos en las aplicaciones que hayamos desarrollado.

La programación orientada a objetos integra las estructuras de datos y las funciones o métodos asociadas a ellas. Las funcionalidades se traducen en colaboraciones entre objetos que se realizan dinámicamente y las estructuras del programa no se ven amenazadas por un cambio en el mismo.

Uno de los principales motivos de utilizar programación orientada a objetos es la adecuación de este modelo a la programación de videojuegos. Podemos distinguir perfectamente los objetos que componen el videojuego y las funciones o capacidades asociadas a ellos los que nos permite que este enfoque sea perfecto para crear una abstracción mediante clases del problema.

Además utilizamos programación orientada a objetos por:

- La orientación a objetos se aproxima más a la forma de pensar de las personas. Esto lo hace más comprensible y fácil de aplicar.
- Proporciona abstracción, ya que en cada momento se consideran sólo los elementos que nos interesan descartando los demás.

- Aumenta la consistencia interna al tratar los atributos y las operaciones como un todo.
- Permite expresar características comunes sin repetir la información.
- Facilita la reutilización de diseños y códigos.
- Facilita la revisión y modificación de los sistemas desarrollados.
- Origina sistemas más estables y robustos.
- Se ha empleado con éxito para desarrollar aplicaciones tan diversas como compiladores, interfaces de usuario, sistemas de gestión de bases de datos, simuladores y, sobre todo, videojuegos y simulaciones.

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos, entre los que encontramos: Smalltalk, Java, C++, Objective-C, PHP, JavaScript,...

4.1.1 Conceptos de la orientación a objetos

Antes de nada se deben conocer ciertos términos y conceptos que se usarán a lo largo de este tema y los siguientes.

- **Clases:** Definiciones de las propiedades y comportamiento de un tipo de objeto concreto (mesa, persona, enemigo en un juego, ...). Se puede asemejar a lo que sería el concepto o el “molde” con el que se crean los objetos de una clase.
- **Atributos**: Los atributos (propiedades) de un objeto son aquellas variables que definen dicho objeto. Las funciones de la clase deben de permitir trabajar con ellos para realizar las modificaciones pertinentes o simplemente para ser consultados ya que son la parte fundamental de dicho objeto.
- **Métodos**: Son cada una de las funcionalidades (comportamiento) que nos aporta una clase para trabajar sobre tipos de datos como pueden ser sus atributos.
- **Instancia:** La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas. Cuando hablamos de una instanciación es el caso en el que a partir de una clase creamos objetos y lo incializamos, llamamos a este paso instanciar una clase.
- **Objetos:** Un objeto es una instancia de una clase o lo que es lo mismo, es una clase inicializada. El objeto está próximo al mundo real mientras que la clase es una abstracción de éste.

4.1.2 Características de la programación orientada a objetos

Vamos a presentar brevemente las características de la orientación a objetos:

- **Abstracción**: Las clases son la abstracción de un conjunto de objetos del mismo tipo. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. El proceso de abstracción permite seleccionar las características relevantes dentro de

un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

- **Encapsulamiento**: Permite proteger y englobar a los datos y la funcionalidad de dichos datos en una estructura, en el mismo nivel de abstracción, que nos permite tenerlos asociados. Esto permite aumentar la cohesión de los componentes del sistema. Generalmente este se suele confundir con el principio de ocultación, principalmente porque son conceptos que suelen ir de la mano.
- **Ocultación de datos**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas.
- **Composición**: Hablamos de composición cuando podemos llegar a componer una clase, a parte de su propio contenido, también con instancias de otros objetos.
- **Generalización (Herencia)**: Una clase puede abstraerse en una clase más general y viceversa. Esto nos permite tener varios tipos de clases hijas de una clase madre según un discriminante con un comportamiento común o individual. Cuando una clase hereda de varias lo llamaremos herencia múltiple.
- **Polimorfismo**: El término viene acuñado de poli (muchos) y morfo (forma), lo cual nos indica que es la facultad de adquirir diferentes formas. Nos permite funciones con el mismo nombre que se diferencien, por ejemplo, en el tipo de parámetros, que tengan comportamientos totalmente diferentes. La forma se adquiere en tiempo de ejecución llamándose a esto “ligadura dinámica”.

Todas estas propiedades y características de la programación orientada a objetos la hacen ideal para la programación de videojuegos. Este tipo de programación sobre un modelo de proceso de desarrollo en espiral que nos permita realizar un desarrollo incremental y el patrón de diseño UML es todo lo que necesitamos para enfrentarnos a la construcción de un software de calidad.

4.2 Concepto de clase

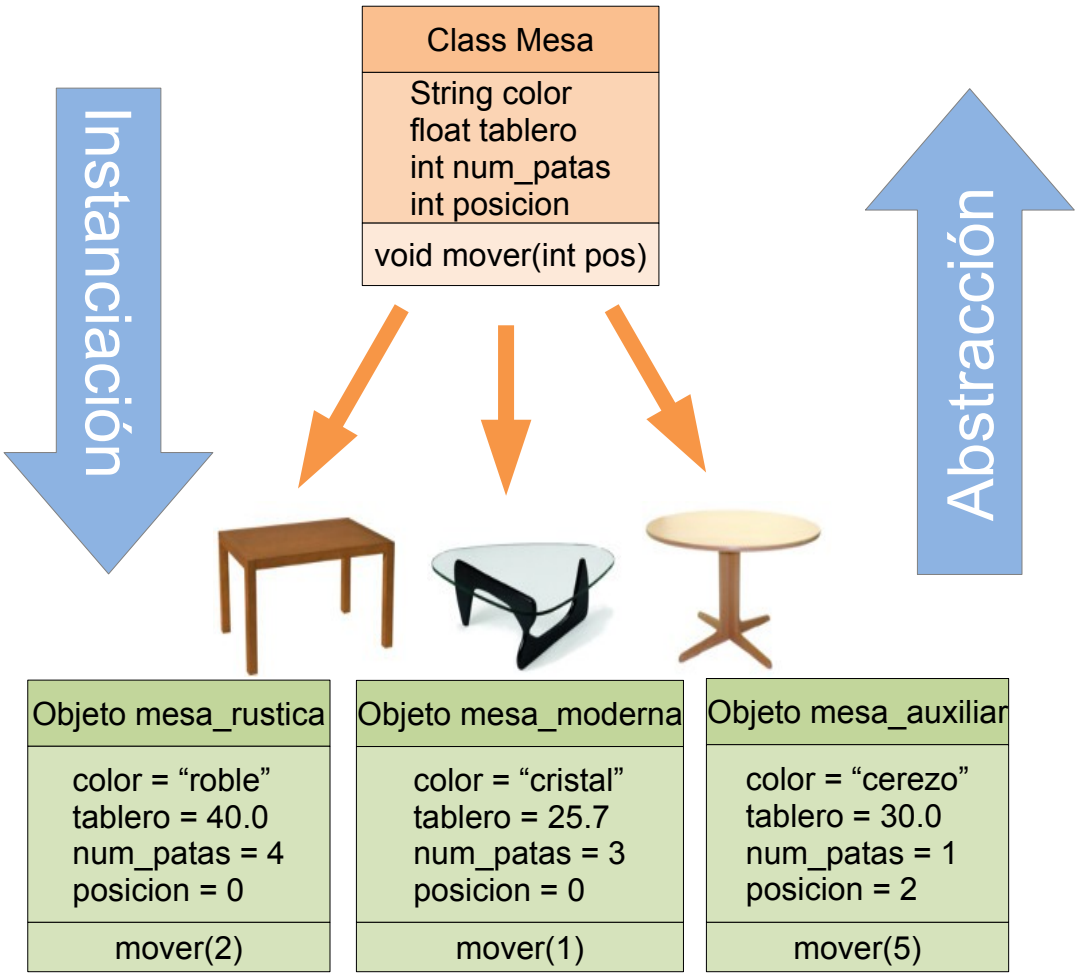
En la programación orientada a objetos, una clase es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. El modelo describe el estado y contiene el comportamiento que todos los objetos creados a partir de esa clase tendrán. Un objeto creado a partir de una determinada clase se denomina una instancia de esa clase.

Una clase por lo general representa un sustantivo, como una persona, lugar o cosa. Es el modelo de un concepto dentro de un programa de computadora. Fundamentalmente, delimita los posibles estados y define el comportamiento del concepto que representa. Encapsula el estado a través de espacios de almacenaje de datos llamados **atributos**, y encapsula el comportamiento a través de secciones de código reutilizables llamadas

métodos.

Más técnicamente, una clase es un conjunto coherente que consiste en un tipo particular de metadatos. Una clase tiene una interfaz y una estructura. La interfaz describe cómo interactuar con la clase y sus instancias con métodos, mientras que la estructura describe cómo los datos se dividen en atributos dentro de una instancia.

Para poder ilustrar todo lo visto hasta ahora se va a proponer el siguiente ejemplo: Todos conocemos el concepto de mesa, nos muestran diferentes mesas y aunque tengan características diferentes sabemos que es una mesa. Pues básicamente vamos a imaginarnos que mediante un proceso de abstracción sacamos aquellas características que hacen que una mesa sea una mesa. Una mesa tiene un tablero, un color, y unas patas para sostenerse, este es el concepto más básico que podemos obtener de mesa, como sabemos existen muchos más tipos de mesas, pero todas básicamente comparten esas características.

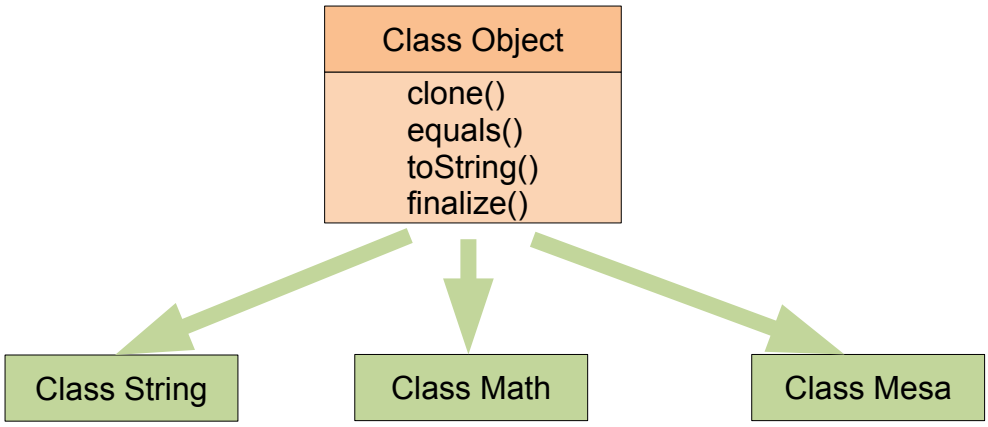


La imagen superior nos aporta mucha información sobre los procesos vistos hasta ahora. Vemos que partimos de una clase mesa, que tiene unos determinados atributos que son el color, el tamaño o superficie del tablero, un número de patas, y una posición, además también como hemos vistos las clases van a definir una serie de acciones o funcionalidades llamadas métodos que son propias de esa clase, para el ejemplo hemos definido el método mover al cual le pasamos como parámetro un número entero que es la nueva posición que va a tener la mesa. Como se puede observar en la imagen para llegar

a la clase mesa realizamos un proceso de abstracción, mientras que para crear cada uno de los objetos particulares realizamos un proceso de instanciación de la clase mesa. En el ejemplo vemos como se crean 3 mesas muy diferentes, cada instancia tiene su espacio en memoria, y cada una de ellas tiene las mismas propiedades que la otra pero con valores diferentes, de tal manera que obtenemos mesas distintas. Así mismo vemos como hemos usado el método mover de la clase mesa para mover cada instancia de la mesa, de tal manera que la "mesa_rustica" al principio su posición era 0, y cuando ejecutamos el método mover(2) sobre ella la posición ya no va a ser 0, sino que será 2.

4.3 La clase Object

Cualquier clase implementada en Java siempre va a ser una subclase de la clase **Object**, esto quiere decir que el origen de todas las clases en Java siempre parten de esta clase, por tanto siempre vamos a tener asociados a nuestras clases como mínimo todos los métodos de la clase Object.



Por tanto todas las clases que provienen (heredan) de Object como mínimo van a tener sus métodos, a parte de los vistos en la imagen hay otros, pero simplemente vamos a dar una explicación básica de los que vemos aquí.

Método	Descripción
clone()	Permite "clonar" un objeto.
equals()	Permite comparar un objeto con otro.
toString()	Devuelve el nombre de la clase.
finalize()	Método invocado por el recolector de basura (garbage collector) para borrar definitivamente un objeto.

4.4 Estructura básica y miembros de una clase

En este apartado nos centraremos sobretodo en conocer cuales son los miembros principales de una clase, pero no solo eso, sino que debemos diferencias entre métodos de clase y de instancia, y para ello introduciremos un nuevo concepto "static".

Para empezar debemos identificar aquellos elementos que pertenecen a una clase, y que ya hemos visto en varias ocasiones a lo largo de este capítulo, se trata de los atributos y métodos. A parte de eso, como veremos en el siguiente apartado tenemos un

"método" muy especial que suele llamarse "constructor", que nos sirve para inicializar un objeto, lo que caracteriza al constructor es que siempre debe llamarse igual que la clase. Vamos a exponer un ejemplo de como sería la clase "mesa".

```
public class Mesa {

	// ***** Atributos *****
	String color;
	float tablero;
	int num_patas;
	int posicion;

	// ***** El constructor *****
	Mesa() {
		// Aquí ponemos el código perteneciente al constructor
	}

	// ***** Los métodos *****
	void mover(int pos) {
		// Aquí ponemos el código perteneciente al método
	}
}
```

Ahí tenemos el código para definir la clase mesa. Evidentemente falta código en el constructor y los métodos, pero aquí lo que nos interesa es sobretodo fijarnos en la estructura. Vemos que lo primero que situamos son los atributos, después el constructor que puede tener o no tener código, y a continuación por último los métodos. Siempre vamos a intentar seguir este orden.

4.4.1 Constructores

Antes hemos hablado de métodos como las funciones que nos permite realizar una clase. Pues podríamos decir que los constructores son un caso especial y particular de lo que es un método. Un constructor es aquel instrumento en la clase que nos sirve para instanciarla y poder crear el objeto con unos determinados parámetros. Para ello debemos también diferenciar entre dos términos muy importantes, definición e instanciación, aunque puedan parecer terminos que indícan lo mismo, esto no es así. Cuando definimos un objeto estamos indicando a que clase va a pertenecer, cuando instanciamos un objeto es cuando lo creamos a través del constructor, esto implica que el sistema le asigna un espacio en memoria para su uso. Veamos un ejemplo:

```
Mesa mesa_roble;      // Definimos el objeto mesa_roble
mesa_roble = new Mesa(); // Instanciamos el objeto mesa_roble
```

En el primer paso vemos que se define el objeto "*mesa_roble*", esto solo implica saber que "mesa_roble" va a ser un objeto de la clase "Mesa", pero aún no se ha creado, por tanto no tiene un espacio ni una referencia a memoria. Sólo en el segundo paso cuando instanciamos el objeto a partir del constructor de "Mesa" se crea un espacio en memoria para el objeto "mesa_roble". Tanto un paso como otro son necesarios para poder crear un objeto, solo que generalmente estos pasos los vemos sintentizados en una sola línea como vemos abajo:

```
Mesa mesa_roble = new Mesa();
```

En el apartado anterior podemos ver como hemos creado un constructor de mesa vacío, y tampoco recibe parámetros. Pero esto en la mayoría de casos no va a ser así. Generalmente nos va a interesar inicializar ciertos variables de nuestra clase. En el caso de la mesa por ejemplo nos puede interesar que el usuario pase como parámetros los valores que va a tener la mesa que va a crear. Llamamos parámetros a aquellos valores que se pasan entre paréntesis cuando llamamos al constructor o un método. Veamos un ejemplo de como quedaría la clase "Mesa" con el constructor con parámetros:

```
public class Mesa {

	// ***** Atributos *****
	String color;
	float tablero;
	int num_patas;
	int posicion;

	// ***** El constructor *****
	Mesa(String c,float tab,int n_patas,int pos) {
		// Aquí ponemos el código perteneciente al constructor
		color=c;
		tablero=tab;
		num_patas=n_patas;
		posicion=pos;
	}

	// ***** Los métodos *****
	void mover(int pos) {
		// Aquí ponemos el código perteneciente al método
	}
}
```

Como podemos observar hay cambios sustanciales respecto al constructor del apartado anterior. Lo primero que podemos observar es que esta vez entre paréntesis se han puesto una serie de "variables", esto son lo que llamamos parámetros, que como vemos cada uno de ellos tiene un tipo y un nombre. Los parámetros son variables a través de los cuales el usuario o el objeto nos va a proporcionar valores que se van a usar en el código del método o constructor. Estos valores solo van a ser válidos durante la ejecución del método o constructor, una vez salimos de este no podrémos usarlos.

Vemos que los parámetros que le pasamos al constructor coinciden en tipo con los atributos de la clase "Mesa". Si analizamos el código interior, lo que vemos también es que cada uno de los atributos de la clase "Mesa" lo igualo a su correspondiente parámetro.

Ahora la pregunta importante debería ser como usamos esto para crear nuestros objetos. En el siguiente código podemos verlo más claro:

```
public class main_Mesa {
	public static void main(String[] args) {
		Mesa mesa_rustica = new Mesa("verde",4.3f,4,0);
		Mesa mesa_moderna = new Mesa("azul",2.4f,1,50);
```

```
    }
}
```

Como podemos observar esto nos ha servido para poder crear o instanciar los objetos "mesa_rustica" y "mesa_moderna" con unos atributos particulares, de tal manera que por ejemplo hemos creado un objeto "mesa_rustica" de color "verde" con un tablero de "4.3" de "4" patas y en la posición "0".

Para poder entender bien el uso y proceso de los parámetros vamos a hacer un seguimiento de lo que ha sucedido con el atributo "color" al crear la "mesa_rustica". Pasámos como parámetro el valor "verde" al constructor, si nos fijamos en el constructor en esa posición está la definición "String c":

```
Mesa(String c,float tab,int n_patas,int pos)
```

Esto lo que provoca es que "c" ahora tenga el valor "verde", si seguimos observando que sucede en el constructor vemos que la siguiente instrucción es:

```
color=c;
```

El atributo "color" hace referencia al atributo de la clase "Mesa", y lo que hacemos es igualar este atributo al valor de lo que le pasamos como parámetro, con lo cual lo que estamos haciendo es:

color = c = "verde";

Por tanto el atributo "color" de la clase pasa a ser "verde". Así mismo ocurre con el resto de parámetros que se le pasan al constructor.

Otro detalle importante es observar que los parámetros se pasan en el mismo orden en el que están definidos en el constructor.

4.4.2 Métodos y atributos de instancia

Cuando instanciamos una clase y creamos un objeto como es el caso de "mesa_rustica", tenemos acceso a todos los atributos y métodos definidos en la clase "Mesa" (a parte también de los que nos ofrece la clase "Object"). Para poder acceder a los atributos y métodos de una clase lo hacemos mediante el "." y seguido el nombre del atributo o el método. De tal manera que por ejemplo podemos hacer esto:

```
public class main_Mesa {
      public static void main(String[] args) {
            Mesa mesa_rustica = new Mesa("verde",4.3f,4,0);
            System.out.println(mesa_rustica.color);
            mesa_rustica.num_patas=2;
      }
}
```

Lo que estamos haciendo por ejemplo con "mesa_rustica.color" es acceder a su atributo color y mostrarlo por pantalla. En el segundo caso con "mesa_rustica.num_patas=2" lo que estamos haciendo es modificando el valor de ese atributo, de tal manera que si antes tenía 4 patas ahora tiene 2 patas.

Como vemos la definición y modificación de atributos de una clase es sencilla, pero también debemos saber definir y usar los métodos.

La sintaxis general para definir un método es:

ámbito tipo_devuelto nombre_método (parámetros)

El “ámbito” hace referencia a la visibilidad del método que puede ser public, private o protected, en este apartado no se va a entrar en estos detalles, de tal manera que nosotros no vamos a usar ninguno hasta que se diga lo contrario, ya que este dato no es necesario indicarlo.

El “tipo_devuelto” hace referencia a qué es lo que devuelve ese método, si al finalizar nos devuelve algún valor, en caso de que nos devuelva algún valor lo indicaremos con el tipo devuelto (por ejemplo int, boolean,... o un objeto incluso como podría ser String). En el caso de que nuestro método no devuelva ningún valor usaremos “void”.

El “nombre_método” no es más que el nombre con el que nosotros queremos llamar a ese método, generalmente usaremos un nombre que sintetice la acción que este realice o que sea claro e intuitivo.

Los “parámetros” son aquellos valores que podemos pasar al método para que sean usados tal y como ocurría en los constructores.

Para que esto se pueda ver más claro veámoslo con un ejemplo, vamos a ir completando la clase “Mesa”:

```
public class Mesa {

	// ***** Atributos *****
	String color;
	float tablero;
	int num_patas;
	int posicion;

	// ***** El constructor *****
	Mesa(String c,float tab,int n_patas,int pos) {
		// Aquí ponemos el código perteneciente al constructor
		color=c;
		tablero=tab;
		num_patas=n_patas;
		posicion=pos;
	}

	// ***** Los métodos *****
	void mover(int pos) {
		// Aquí ponemos el código perteneciente al método
		posicion=pos;
	}
	int devolverPosicion() {
		return posicion;
	}
}
```

Como novedad hemos introducido los métodos “mover” y “devolverPosicion”. El primer método lo que hace es cambiar el valor del atributo “posicion” de la instancia pos el nuevo valor que le pasamos como parámetro y que llamamos “pos”. Como este método no devuelve ningún valor vemos que usamos “void”, el nombre del método es “mover” y los parámetros en este caso sería “int pos”.

El segundo método parece tener una estructura diferente, y es que en este caso si devolvemos un valor, devolveriamos el valor del atributo “posicion” que en este caso es de tipo “int”. Un dato importante es que cuando nuestro método devuelve un valor, el valor que devolvemos debemos ponerlo en el código precedido de la palabra “return”, si nos fijamos en el código devolvemos el valor del atributo “posicion” y lo hacemos con “return”.

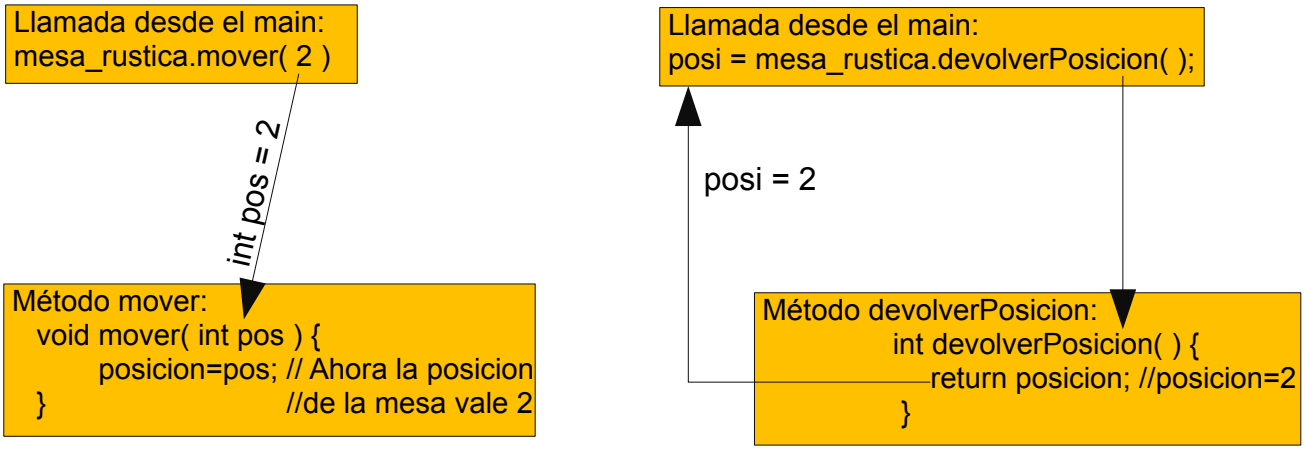
Ahora veamos como usamos estos atributos en nuestra clase “Main”:

```
public class main_Mesa {
  public static void main(String[] args) {
      int posi=-1;
      Mesa mesa_rustica = new Mesa("verde",4.3f,4,0);
      posi = mesa_rustica.devolverPosicion();
      System.out.println("La pos:"+posi);
      mesa_rustica.mover(2);
      posi = mesa_rustica.devolverPosicion();
      System.out.println("Nueva pos:"+posi);
  }
}
```

El resultado mostrado por consola al ejecutarlo sería:

```
La pos:0
Nueva pos:2
```

Vamos a analizar lo que ha sucedido de una manera más gráfica:



4.4.3 Referencia al objeto “this”

El objeto “this” hace referencia al contexto de la clase, es la referencia directa a la clase. Es decir, cuando usamos dentro de la clase “this” podemos referenciar todo lo que en ella está contenido, ya que es una referencia a ella misma. Por tanto si yo escribo “this” y a continuación un punto “.”, podremos acceder a todos los atributos y métodos de la clase.

Quizá la utilidad de esto puede quedarse un poco parca, pero si lo vemos con un sencillo ejemplo podremos ver su utilidad.

Vamos a revisar el constructor de “Mesa” y en vez de llamarle al parámetro que hace referencia al color como “String c”, vamos a llamarlo “String color”, o sea que va a tener el mismo nombre que el atributo de la clase, entonces nos quedaría de la siguiente manera:

```
public class Mesa {

	// ***** Atributos *****
	String color;
	float tablero;
	int num_patas;
	int posicion;

	// ***** El constructor *****
	Mesa(String color,float tab,int n_patas,int pos) {
		// Aquí ponemos el código perteneciente al constructor
		color=color;
		tablero=tab;
		num_patas=n_patas;
		posicion=pos;
	}
}
```

Como se puede observar ahora el atributo de la clase "color" y el parámetro "color" se llaman igual, esto produce un problema, y es que el compilador no distingue cual es cual, así que la manera de hacerle saber al compilador cuál es el que hace referencia al atributo de la clase es haciendo esto "this.color=color". Además si nos fijamos, en eclipse antes nos aparecía "color" en ambos lados en negro, esto es porque está tomando el parámetro al no distinguirlos, en cambio si hiciese referencia al atributo de la clase lo pondría de color azul. De tal manera que en vez de "color=color", lo que tenemos que poner es:

```
this.color=color;
```

Ejercicio de auto-evaluación A4.1: *Crea una clase coche que incluya al menos los atributos modelo, número de ruedas, un depósito de gasolina y un atributo para saber si está encendido o apagado. Debe crearse un constructor y contemplarse al menos los métodos encender y apagar, otro para mostrar los datos del coche, y un método para correr que dependerá de si el coche está encendido o apagado para que funcione, este metodo correr decrementará el depósito de gasolina en uno cada vez que sea invocado, y si el depósito llega a cero debe apagar el coche. El coche sin gasolina no se puede encender.*

4.4.4 Métodos y atributos de clase (static)

Hasta el momento en las clases los atributos y métodos que hemos creado solo nos sirven para ser llamados una vez hemos instanciado el objeto, como por ejemplo la posición de la mesa y el método mover solo puedo usarlos una vez he creado una mesa, por tanto son atributos y métodos que pertencen a las instancias.

En este apartado veremos que también existen los atributos y métodos de la clase. Esto quiere decir que son atributos y métodos que podemos invocar sin necesidad de que se haya creado una instancia, sino que podemos acceder a ellos a través de la clase

directamente. No obstante estos atributos y métodos también pueden ser accedidos desde la instancia, pero se debe tener en cuenta que la repercusión de los cambios no está limitada al ámbito de la instancia, sino al de la clase.

Los atributos y métodos de clase se identifican por la palabra reservada **static.** Un detalle importante a tener en cuenta es que desde un método estático solo podemos hacer referencia a otros atributos y métodos estáticos. Esto tiene sentido, ya que si intentasemos llamar a un método o atributo de una instancia desde una clase sería imposible.

El motivo por el cual podemos querer usar atributos o métodos estáticos es porque puede que necesitemos que ciertos parámetros o funciones de las instancias sean comunes a todas ellas y compartidos por las mismas, siempre teniendo en cuenta que un cambio sobre un atributo estático va a repercutir sobre el resto de instancias. Esto también implica un ahorro de memoria, ya que si todas las instancias deben compartir un atributo que tiene que tener el mismo valor para todas, o un método que realiza la misma función sobre atributos estáticos en todas ellas, solo vamos a tener un segmento de memoria ocupado para cualquier atributo o método estático aunque tengamos varias instancias.

Vamos a suponer que tenemos una empresa que pone a disposición de sus clientes unas cuentas y de las cuales saca un beneficio anual en base al procentaje de la tasa de anual equivalente (TAE). El código para representar este esquema sería el siguiente:

```
public class cuenta {
	static String nombreEmpresa;
	static int capital=0;
	static float beneficio=0;
	int saldo;
	String titular;

	public cuenta(String titular,int saldo){
		this.titular=titular;
		this.saldo=saldo;
		capital=capital+saldo;
	}

	public static void ponerNombreEmpresa(String nombre){
		nombreEmpresa=nombre;
	}

	public void mostrarCapitales(){
		System.out.println("El saldo "+titular+" es:"+saldo);
		System.out.println("El capital de la empresa "+nombreEmpresa+"
es:"+(capital+beneficio));
	}

	public static float beneficioAnual(float TAE){
		beneficio=beneficio+capital*TAE;
		return beneficio;
	}
```

```
}
```

Como podemos observar se ha creado un atributo estático para el nombre de la empresa, ya que ese nombre va a ser el mismo en todas las cuentas. El beneficio de la empresa también es el mismo en todas, y el capital del que dispone la empresa igualmente. El saldo es el único atributo que va a pertenecer a la instancia, ya que cada cuenta va a tener un saldo diferente.

Vamos a crear dos cuentas y ver que sucede en cada uno de los casos:

```
public class cuentaMain {
	public static void main(String[] args) {
		cuenta.ponerNombreEmpresa("Empresa Andrés");
		cuenta pepe=new cuenta("Pepe",50);
		cuenta maria=new cuenta("María",125);
		// Mostramos los datos
		pepe.mostrarCapitales();
		maria.mostrarCapitales();
		// Calculamos el beneficio a través de la clase cuenta
		System.out.println("Beneficio
			(clase):"+cuenta.beneficioAnual(0.1f));
		// A través de la instancia pepe cambiamos el nombre
		pepe.ponerNombreEmpresa("Empresa Pepe");
		// A través de maria vemos los cambios realizados
		maria.mostrarCapitales();
		// La instancia maria también puede llamar a los beneficios
		System.out.println("Beneficio
			(instancia):"+maria.beneficioAnual(0.2f));
		// La instancia pepe visualiza los datos
		pepe.mostrarCapitales();
	}
}
```

Los resultados obtenidos ejecutando el código mostrado son:

```
El saldo Pepe es:50
El capital de la empresa Empresa Andrés es:175.0
El saldo María es:125
El capital de la empresa Empresa Andrés es:175.0
Beneficio (clase):17.5
El saldo María es:125
El capital de la empresa Empresa Pepe es:192.5
Beneficio (instancia):52.5
El saldo Pepe es:50
El capital de la empresa Empresa Pepe es:227.5
```

Primero ponemos nombre a la empres a través de la clase cuenta. Después creamos dos cuentas, “pepe” y “maria”. Al visualizar los capitales en ambas instancias vemos que el capital de la empresa es el mismo, ya que es un atributo de la clase. También podemos observar como calculamos los beneficios anuales a través de la clase, ya que el método

es estático. Pero de igual forma como se ha comentado los atributos y métodos que no son estáticos no son accesibles desde la clase, pero si al contrario, los métodos y atributos estáticos pueden ser accedidos desde las instancias, por tanto vemos como "maria" también puede calcular los beneficios y "pepe" puede cambiar el nombre de la empresa, y estos datos se van a ver reflejados en todas las instancias.

Veamos otro ejemplo, supongamos que queremos usar un método que pueda invocarse directamente desde el main, tendríamos que hacerlo de la siguiente manera:

```
public class pedirNombreMain {
	static String pedirNombre(){
		String nombre;
		Scanner sc=new Scanner(System.in);
		System.out.print("Dame tu nombre:");
		nombre=sc.next();
		return nombre;
	}
	public static void main(String[] args) {
		String nombre;
		nombre=pedirNombre();
	}
}
```

Como el método "main" es estático, solo podemos llamar a métodos estáticos, para llamar a un método de una instancia deberíamos antes crear la instancia.

Ejercicio de auto-evaluación A4.2: *¿Porqué el método "main" es estático?. Justifica tu respuesta.*

Ejercicio de auto-evaluación A4.3: *Crea un programa que tenga un "main" que invoca a un método llamado "menu" que nos muestra un menú en pantalla con 3 opciones, siendo la última de estas la opción "salir", mientras que no pulsemos la opción salir el programa seguirá mostrando el menú. Ejemplo del menú:*

MENU
****
1. Opción 1
2. Opción 2
0. Salir

Elige una opción:

Ejercicio de auto-evaluación A4.4: *Crea una clase familia. Los miembros de una familia van a tener un nombre, un apellido, una dirección y una edad. Todos los miembros de la familia van a tener el mismo apellido, y además cuando se cambie su dirección de residencia va a cambiar la de todos los miembros de la familia. Se pide al menos crear un constructor, un método para mostrar los datos del miembro de la familia, un método para cambiar la dirección de la familia y un método para poner apellido a la familia. Crear al menos 4 personas en la familia.*

4.4.5 Atributos locales y globales de clase.

En este apartado hablaremos del ámbito o límites del alcance que tiene una atributo en la clase. Debemos distinguir dos ámbitos bien diferenciados en la clase, los atributos que tienen un ámbito global, que son aquellos que se definen al principio de la clase, son sus atributos propiamente dichos, y que pueden ser accedidos desde todos los métodos de la clase, y después tenemos los que tienen un ámbito local, que son aquellos que son creados dentro del ámbito de un método y cuando se invoca el método se crean en memoria y cuando el método acaba se destruyen de la memoria.

En el siguiente ejemplo se pueden ver diferenciados ambos:

```
public class localglobal {
  int numero;

  public localglobal(int num){
    numero=num;
  }

  public void mostrarGlobal(){
    System.out.println("El valor de número
      es:"+numero);
  }

  public void mostrarLocal(){
    int numero=-90;
    System.out.println("El valor de número
      es:"+numero+" el valor de número global
       es:"+this.numero);
  }
}
```

Atributo global de la clase

En el constructor podemos ver Un ejemplo de variable global "numero" y local "num".

Aquí se muestra el valor del Atributo global

Aquí como vemos tenemos otro atributo que se llama igual que el atributo global de la clase, por tanto si usamos el identificador numero hacemos referencia a la variable local si queremos hacer referencia a la global necesitamos usar "this".

Como se puede observar en el código anterior usamos dos atributos que se llaman igual, solo que uno es global y el otro es local, la diferencia es que cuando creamos una instancia, el atributo global "numero" tiene un espacio en memoria destinado hasta que la instancia se destruye, y puede ser accedido desde todos los métodos de la clase, en cambio el atributo local "numero" del método "mostrarLocal()" solo puede ser accedido desde él, y se crea en memoria solo cuando invocamos a ese metodo, y cuando acaba se destuye. También se puede observar como los parámetros de los métodos siempre van a ser variables o atributos locales.

Si tenemos el siguiente código en un "main" y lo ejecutamos:

```
public class localglobalMain {
      public static void main(String[] args) {
            localglobal lg=new localglobal(5);

            lg.mostrarGlobal();
            lg.mostrarLocal();
      }
}
```

El resultado será:

```
El valor de número es:5
El valor de número es:-90 el valor de número global es:5
```

4.4.6 Paso de parámetros por valor y por referencia

Como hemos visto en apartados anteriores los métodos tienen tanto parámetros, como un posible valor que pueden devolvernos. Los parámetros cuando son pasados una vez acaba el método se limpian de la memoria, por tanto no conservan su valor, si queremos que estos cambios fuesen permanentes hasta ahora lo hemos hecho con el valor que devuelve la función, con el "return". Mediante el paso de parámetros por referencia podemos hacer que los cambios sean permanentes no solo con un valor, sino con varios. Antes de continuar vamos a definir que es el paso de parámetros por valor y por referencia.

Paso de parámetros por valor: las variables que son pasadas al método como parámetros solo copian su valor, no son modificadas. Vamos a ilustrarlo:

Supongamos el siguiente código:

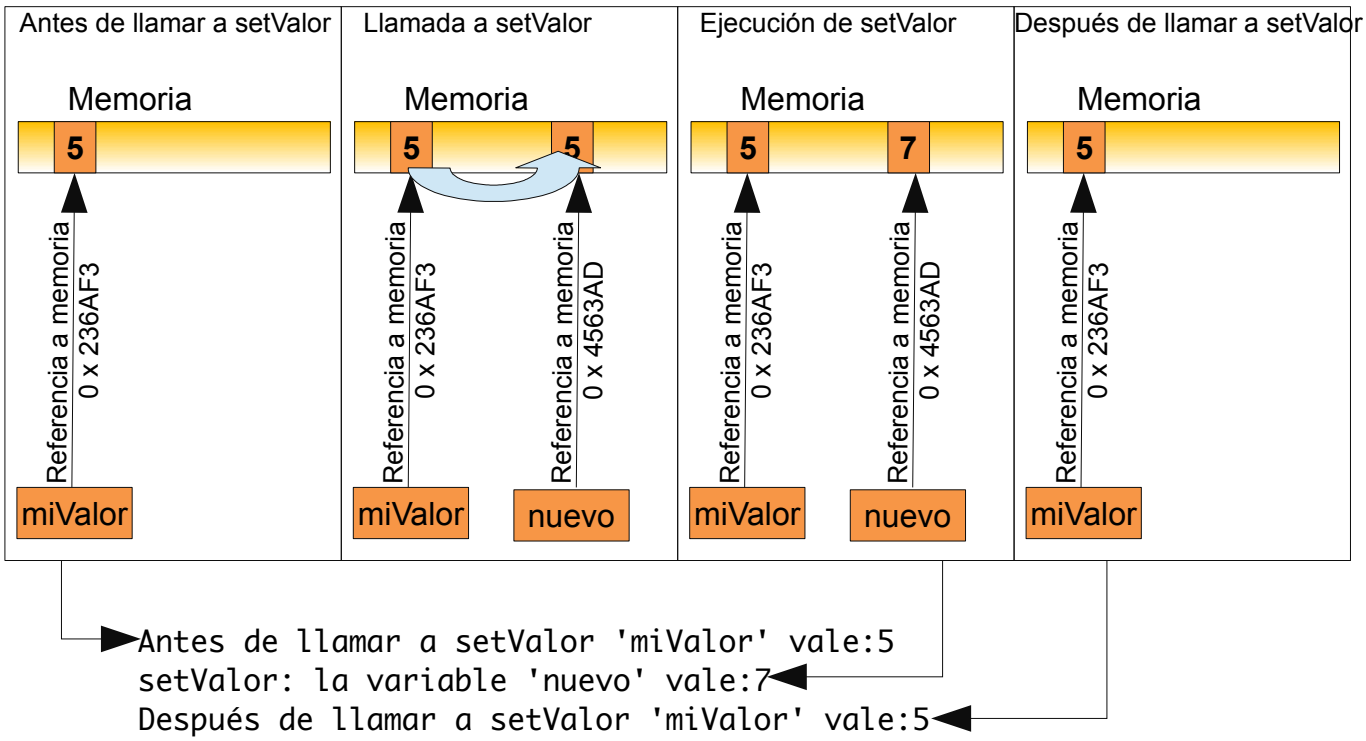
```
public class pasoValor {
	public static void setValor(int nuevo){
		nuevo = nuevo + 2;
		System.out.println("setValor: la variable 'nuevo'
			vale:"+nuevo);
	}
	public static void main(String[] args){
		int miValor = 5;
		System.out.println("Antes de llamar a setValor 'miValor'
			vale:"+miValor);
		setValor(miValor);
		System.out.println("Después de llamar a setValor 'miValor'
			vale:"+miValor);
	}
}
```

El resultado de ese código sería:

```
		Antes de llamar a setValor 'miValor' vale:5
		setValor: la variable 'nuevo' vale:7
		Después de llamar a setValor 'miValor' vale:5
```

Como vemos el valor de "miValor", no cambia.

Veamos gráficamente que ha sucedido:



Paso de parámetros por referencia: las variables que son pasadas al método no copian su valor, lo que pasan es la dirección de memoria en la que están alojadas.

Supongamos que tenemos el siguiente código, por un lado la case mascota:

```
public class mascota {
	public String nombre;
	public mascota(String nombreMascota){
		nombre = nombreMascota;
	}
}
```

Como vemos se ha usado un atributo “public”, esto lo haremos así para una mayor simplicidad del código y que se entienda más fácilmente, pero recordemos que a partir de ahora en adelante siempre que no se especifique lo contrario usaremos atributos privados y proporcionaremos una interfaz de acceso a ellos mediante métodos.

Por otro lado tenemos la clase main para crear una mascota:

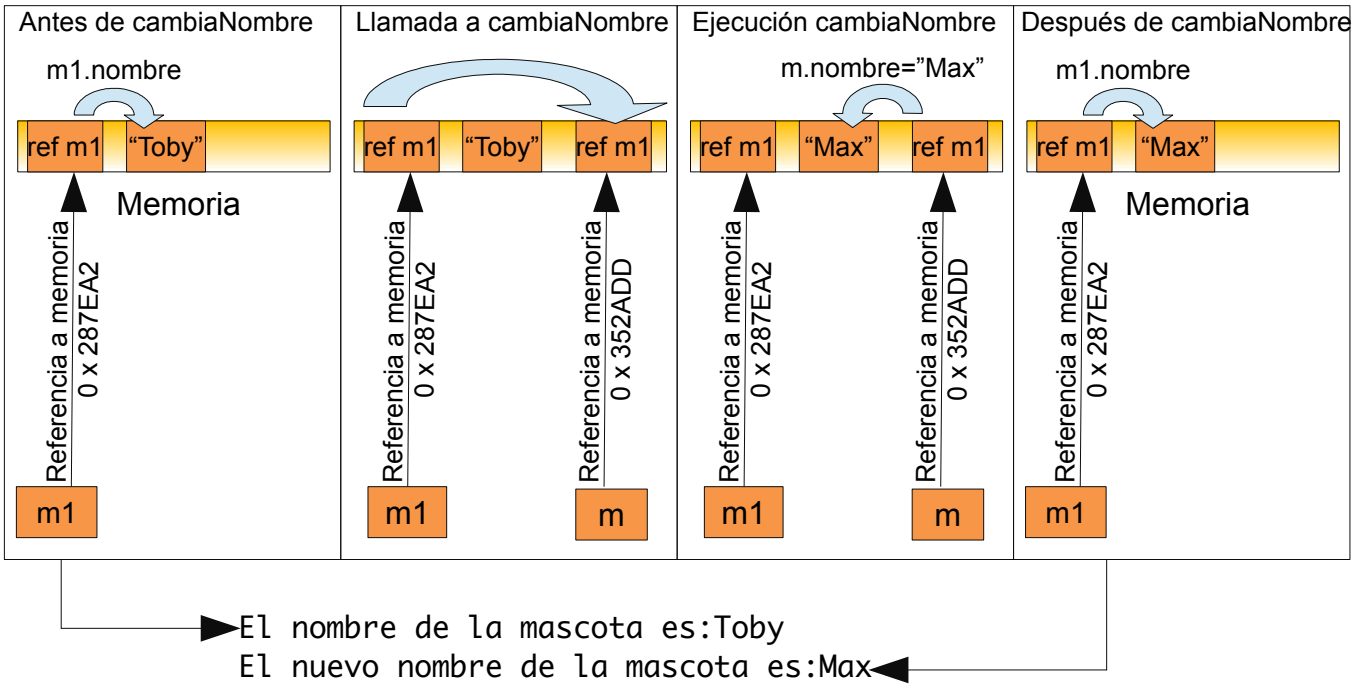
```
public class mainMascota {
	public static void cambiaNombre(mascota m){
		m.nombre = "Max";
	}
	public static void main(String[] args) {
		mascota m1=new mascota("Toby");
		System.out.println("El nombre de la mascota es:"+m1.nombre);
```

```
            cambiaNombre(m1);
            System.out.println("El nuevo nombre de la mascota
                  es:"+m1.nombre);
      }
}
```

Si podecemos a su ejecución el resultado será el siguiente:

```
El nombre de la mascota es:Toby
El nuevo nombre de la mascota es:Max
```

Como vemos el atributo “nombre” de la mascota “m1” ha cambiado. Veamos gráficamente que ha sucedido:



Generalmente en Java se tiene entendido que los tipos básicos son elementos que siempre se pasan por valor, y los objetos son los que se pasan por referencia, exceptuando casos particulares de objetos inmutables como String, Integer, Double, etc... Una cosa importante que debemos tener en cuenta es que en Java, aunque nos parezca que el paso de parámetros con objetos es por referencia, siempre se hace un paso de parámetros por valor. Entonces nos podemos preguntar, ¿como es posible que se cambien los valores?, la respuesta es simplemente porque lo que se hace es copiar la referencia del objeto que hay en memoria, la referencia no se cambia, por eso es una copia, pero lo que cambiemos en ese objeto se queda permanentemente.

Tal y como hemos dicho esto no sucede con los objetos inmutables, probemos un ejemplo con la clase “Integer”:

```
public class inmutableMain {
      public static void cambiaValor(Integer i){
            i=i*2;
```

```
            System.out.println("El valor dentro de cambiaValor es:"+i);
      }
      public static void main(String[] args) {
            Integer miValor=new Integer(5);
            System.out.println("El valor de 'miValor' es:"+miValor);
            cambiaValor(miValor);
            System.out.println("Después 'miValor' vale:"+miValor);
      }
}
```

Esto nos da como resultado:

```
El valor de 'miValor' es:5
El valor dentro de cambiaValor es:10
Después 'miValor' vale:5
```

4.4.7 Visibilidad de métodos y atributos

Este apartado se centra principalmente en la encapsulación y la ocultación de información. Los métodos y atributos tienen una visibilidad dependiendo de si son declarados como públicos, privados, protegidos, o no especificamos nada. Principalmente nos vamos a centrar en los que están marcados en amarillo, ya que los marcados en verde debemos abordarlos una vez conozcamos el concepto de “Herencia” en temas posteriores.

Así que según esta tabla podemos observar la visibilidad que tienen según como estén definidos:

Método o Atributo	public	protected	private	Sin especificar
¿Accesible desde la propia clase?	Si	Si	Si	Si
¿Accesible desde otras clases en el mismo paquete?	Si	Si	No	Si
¿Accesible desde una subclase en el mismo paquete?	Si	Si	No	Si
¿Accesible desde subclases en otros paquetes?	Si	(*)	No	No
¿Accesible desde otras clases en otros paquetes?	Si	No	No	No

(*) Se puede acceder por los objetos de la subclase pero no por los de la superclase.

De lo anterior podemos sacar algunas conclusiones y reglas que usaremos a partir de ahora en adelante para la resolución de ejercicios.

La primera es que las clases siempre que no se especifique lo contrario van a ser públicas o sin especificar para poder tener al menos acceso desde el paquete, por tanto siempre empezaremos creando las clases como:

```
public class nombreClase { ... }
```

La segunda es que el acceso a los constructores siempre va a ser público o sin especificar para poder tener al menos acceso desde el paquete, por tanto siempre

empezaremos creando los constructores como:

```
public class nombreClase {
      ...
      public nombreClase(){ ... }
}
```

La tercera es que el ámbito de los atributos debe ser privado siempre que sea posible, por tanto debemos porporcionar siempre una “interfaz de acceso” al atributo cuando sea necesario, para ello usaremos la nomenclatura de **get** (para recoger el valor del atributo) y **set** (para asignar un valor al atributo), por tanto:

```
public class nombreClase {
      private int atributo1;
      ...
      public nombreClase(){ ... }
      public int getAtributo1(){
            ...
            return atributo1;
      }
      public void setAtributo1(int atrib){
            ...
            atributo1 = atrib;
      }
}
```

Por último solo debemos proporcionar acceso a aquellos métodos que van a ser accedidos fuera de la clase, a través de una instancia, aquellos que sean accedidos desde dentro de la clase deberán ser privados para cumplir el principio de ocultación de la información, por ejemplo imaginemos que hay un método que realiza unos calculos de conversión antes de asignar el valor del atributo1 en el set:

```
public class nombreClase {
      private int atributo1;
      ...
      public nombreClase(){ ... }
      public int getAtributo1(){
            ...
            return atributo1;
      }

      private int conversion(int atrib){
            ...
            return atrib;
      }

      public void setAtributo1(int atrib){
            ...
            atributo1 = conversion(atrib);
      }
```

```
}
```

4.4.8 Composición de clases

La composición de clases está basada en aplicar conceptos de modularidad y reutilización. Consiste en realizar nuevas clases a partir de la composición de otras que ya tenemos creadas.

Para ilustrar estos conceptos se va a utilizar un ejemplo sencillo en el que vamos a crear una clase "Piloto" que va a constar de un nombre y altura. Después de esto vamos a crear una clase "Coche" en la cual vamos a meter un conductor que va a ser un piloto. La clase "Coche" va a tener un modelo y conductor y un método para mostrar y cambiar de conductor.

Vamos primero con la clase "Piloto.java":

```
public class Piloto {
	String nombre;
	float altura;
	public Piloto(String nom, float alt){
		nombre=nom;
		altura=alt;
	}

	public void mostrar(){
		System.out.println("Nombre:"+nombre+"\tAltura:"+altura);
	}
}
```

Ahora creamos la clase "Coche.java":

```
public class Coche {
	String modelo;
	Piloto conductor;
```

Esto es la composición

```
	public Coche(String mod, Piloto cond){
		modelo=mod;
		conductor=cond;
	}
```

Como vemos en el constructor le pasamos un Piloto e inicializamos el conductor

```
	public void cambiarConductor(Piloto c){
		conductor=c;
	}
```

Como vemos en el método le pasamos un Piloto y cambiamos el conductor

```
	public void mostrar()
	{
		System.out.println("El coche es:"+modelo+" y su conductor
es:");
		conductor.mostrar();
	}
```

Llamamos al método mostrar que pertenece a la clase Piloto

```
}
```

Y el main sería el siguiente:

```
public static void main(String[] args) {
      Piloto pepe=new Piloto("Pepe",1.9f);
      Piloto juan=new Piloto("Juan",1.75f);
      Coche bmw=new Coche("BMW",pepe);

      pepe.mostrar();
      bmw.mostrar();
      bmw.cambiarConductor(juan);
      bmw.mostrar();
}
```

Esto nos da como resultado:

```
Nombre:Pepe     Altura:1.9
El coche es:BMW y su conductor es:
Nombre:Pepe     Altura:1.9
El coche es:BMW y su conductor es:
Nombre:Juan     Altura:1.75
```

Como vemos lo que hemos hecho es mostrar el conductor “pepe” e introducirlo dentro del coche a través del constructor de coche, al mostrar el coche vemos que el conductor es “pepe”, después introducimos el conductor “juan” dentro del coche mediante el método “cambiarConductor” y mostramos los datos del coche de nuevo, y observamos que el conductor ahora es “juan”.

4.4.9 Métodos recursivos

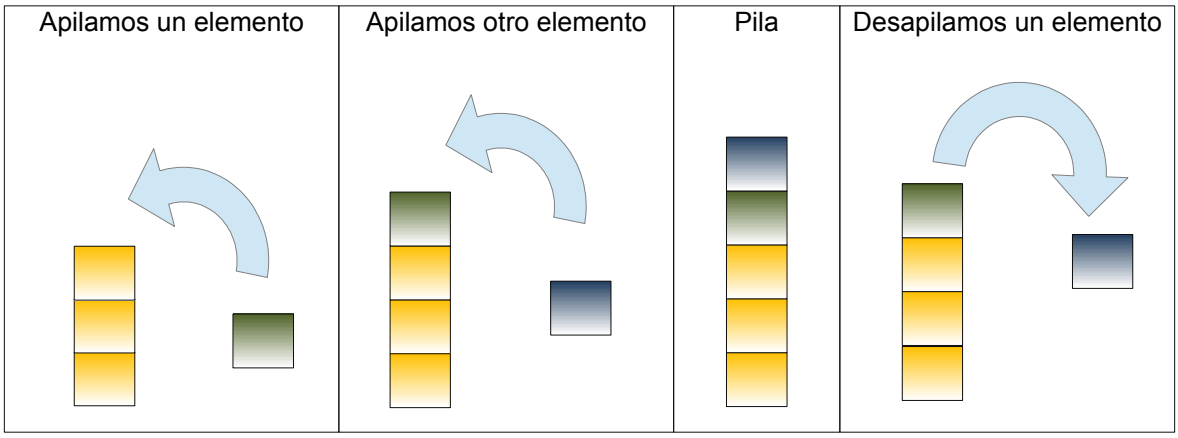
En programación se llama método recursivo a aquel que se llama así mismo. Todo algoritmo recursivo tiene su equivalente en su versión iterativa (con bucles). Por tanto siempre un algoritmo recursivo puede transformarse en iterativo y a la inversa.

Entonces, realmente ¿donde reside la diferencia?. Los algoritmos recursivos son menos eficientes que los iterativos, ocupan mas memoria, ya que se usa una estructura de pila para solucionar las llamadas recursivas. Pero la estructura y la solución al problema son mucho mas claras que una iterativa. De esta manera se suelen usar por ejemplo soluciones recursivas en problemas como laberintos, recorrido de arboes, etc...

Todo algoritmo recursivo debe tener al menos una llamada recursiva, y una llamada que no es recursiva que finaliza la recursión, que a partir de ahora llamaremos caso base, y evita que el algoritmo caiga en un bucle infinito de llamadas a si mismo.

¿Que es una estructura de pila?

Una estructura de pila es aquella en la cual se van apilando elementos y desapilando, de tal manera que el último elemento en entrar sería el primero en salir. Vamos a ver un ejemplo gráfico:



Ejemplo de algoritmo recursivo

Uno de los ejemplos mas representativos de la recursión es la función factorial. El factorial de un número es el resultado de la multiplicación desde ese número hasta que llegamos a 1. La operación se suele representar con el signo de admiración "!", por ejemplo el factorial de 5 sería:

5! = 5x4x3x2x1 = 120

Si esto lo tenemos que solucionar con lo visto hasta ahora, vamos a crear un método que nos resuelve el factorial de manera iterativa, la solución iterativa (con bucles) sería la siguiente:

```
public class mainIterativo {
	public static int factorialIterativo(int num){
		int res=num;
		do{
			res=res*(--num);
		}while(num>1);
		return res;
	}

	public static void main(String[] args) {
		int valor1=5;
		System.out.println("El factorial de "+valor1+" es "
			+factorialIterativo(valor1));
	}
}
```

Como vemos, en el algoritmo iterativo finaliza el bucle "doWhile" cuando el número es cero o menor que cero, por tanto esté va a ser nuestro caso base, el resto es el caso recursivo. Teniendo esto en cuenta veamos la solución recursiva:

```
public class mainRecursivo {
  public static int factorialRecursivo(int num){
    if (num==1){
      return 1;
```

Caso base

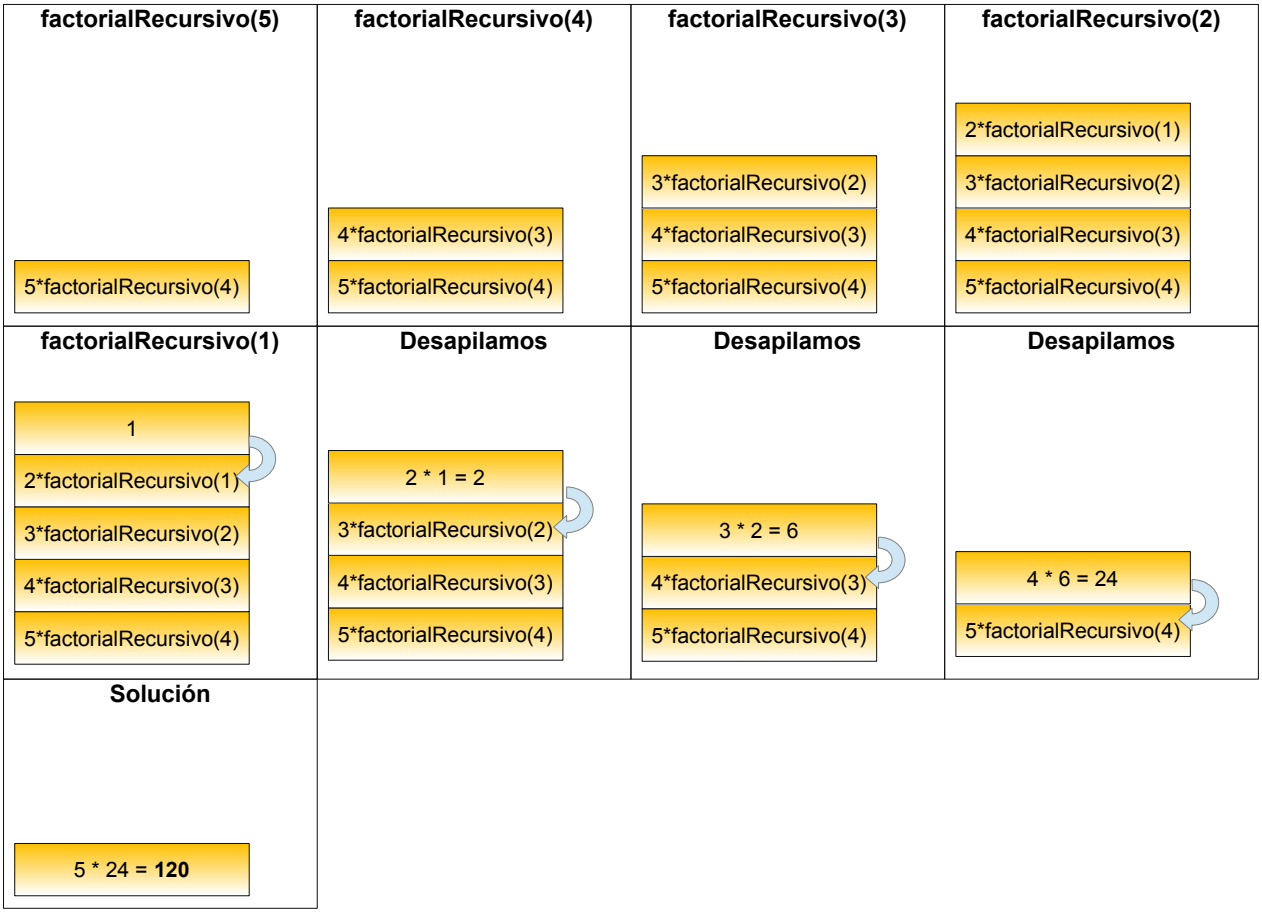
```
    } else {
      return num*factorialRecursivo(--num);
    }
  }

  public static void main(String[] args) {
    int valor1=5;
    System.out.println("El factorial de "+valor1+" es "
     +factorialRecursivo(valor1));
  }
}
```

Caso recursivo

Vamos a analizar paso a paso la ejecución del programa para el factorial de 5:

Si nos fijamos al comenzar num valdrá 5, por tanto hemos de calcular factorialRecursivo(5) y el resultado será num*factorialRecursivo(- - num), lógicamente si sustituimos el valor de num, el resultado es 5*factorialRecursivo(4), el problema es que antes de solucionarlo tenemos que solucionar el factorialRecursivo(4), entonces la operación se queda pendiente en la pila hasta que se solucione. Vamos a verlo de una manera más gráfica:



Ejercicio de auto-evaluación A4.5: *Realizar un programa que calcule la potencia de un número de manera recursiva. Por ejemplo: $2^3 = 8$.*

Ejercicio de auto-evaluación A4.6: *Realizar un programa que dada una palabra nos la muestre al revés usando un algoritmo recursivo. Por ejemplo: "Hola" → "aloH"*

4.5 Sobrecarga

La sobrecarga es una forma que tenemos de realizar diferentes acciones mediante la llamada a un mismo método o constructor por medio de diferentes parámetos. Esto quiere decir que por ejemplo yo puedo tener varios métodos que se llamen igual y que el compilador va a diferenciar en base al número de parámetros o a igual número de parámetros en base al orden de estos.

4.5.1 Sobrecarga de constructores

Deinifmos sobrecarga de constructores en una clase cuando necesitamos que un objeto sea inicializado de diferentes formas. Veamos el siguiente ejemplo con la clase "circulo":

```
public class circulo {
	private int radio;
	private String color;
	public circulo(){
		radio=0;
		color="SIN COLOR";
	}
	public circulo(int radio){
		color="SIN COLOR";
		this.radio=radio;
	}
	public circulo(String color,int radio){
		this.color=color;
		this.radio=radio;
	}
	public circulo(int diametro,String color){
		this.color=color;
		radio=diametro/2;
	}
	public void mostrar(){
		System.out.println("Cículo '"+color+"' de radio:"+radio);
	}
}
```

Como podemos observar tenemos hasta 4 constructores. El primero es el constructor por defecto, el cual no recibe parámetros, por tanto si en un main hacemos lo siguiente:

```
circulo c=new circulo();
```

```
c.mostrar();
```

El resultado será: `Cículo 'SIN COLOR' de radio:0`

Si probamos el segundo constructor:

```
circulo c=new circulo(5);
c.mostrar();
```

El resultado será: `Cículo 'SIN COLOR' de radio:5`

Si probamos el tercer constructor:

```
circulo c=new circulo("ROJO",5);
c.mostrar();
```

El resultado será: `Cículo 'ROJO' de radio:5`

Si probamos el cuarto constructor:

```
circulo c=new circulo(40,"VERDE");
c.mostrar();
```

El resultado será: `Cículo 'VERDE' de radio:20`

Como se puede observar los dos últimos constructores reciben el mismo número de parámetros, entonces, ¿como se diferencian?, pues la respuesta es bien sencilla, aunque tengan el mismo número de parámetros el orden es diferente, en el tercer constructor se le pasa un “String” primero y después un “int”, y en el cuarto constructor se le pasa primero un “int” y después un “String”.

4.5.2 Sobrecarga de métodos

Lo que hemos visto para la sobrecarga de constructores vale igualmente para la sobrecarga de métodos. Podemos tener varios métodos que se llaman igualmente y se van a diferenciar en la cantidad o el orden de los parámetros. Una cosa a tener en cuenta es que el valor devuelto por un mismo método o no, no es razón de diferenciación, esto sucede porque cuando llamamos a un método, aunque devuelva un valor no siempre tiene que ser recogido, entonces el compilador no tendría manera de diferenciarlo respecto a otro método.

Veamos un ejemplo de sobrecarga de métodos:

```
public class calculadora {
      int resultado;
      public calculadora(){
            resultado=0;
      }
      int sumar(int v1,int v2){
            resultado=v1+v2;
            return resultado;
      }
      int sumar(double v1,double v2){
            resultado=(int)(v1+v2);
            return resultado;
      }
      int sumar(int v1,int v2,int v3){
            resultado=v1+v2+v3;
```

```
            return resultado;
        }
    }
```

Si realizamos la siguiente ejecución en un main:

```
    public static void main(String[] args) {
        calculadora c=new calculadora();
        System.out.println("Método sumar(int v1,int
            v2):"+c.sumar(2,3));
        System.out.println("Método sumar(double v1,double
            v2):"+c.sumar(2.3,4.7));
        System.out.println("Método sumar(int v1,int v2,int
            v3):"+c.sumar(2,3,1));
    }
```

Y el resultado sería:

```
Método sumar(int v1,int v2):5
Método sumar(double v1,double v2):7
Método sumar(int v1,int v2,int v3):6
```

4.6 Destrucción de objetos

Cuando estamos ejecutando un programa y ciertos objetos dejan de usarse, ¿que ocurre con ese espacio en memoria?. Por suerte en Java contamos con el Garbage Collector o recolector de basura de Java. Esto es una facilidad que nos proporciona Java para liberar memoria y que funciona de manera transparente al programador y usuario. El recolector de basura de Java está basado en las referencias que existen hacia una posición de memoria, si una determinada posición de memoria está referenciada por ejemplo por 2 objetos, cuando uno de ellos acaba y deja de referenciarla, el recolector de basura aún no puede liberar esa memoria porque queda un objeto aún apuntando a ella, una vez que ese último objeto deja de apuntar a esa posición de memoria seguidamente el recolector de basura actua y libera esa memoria para que pueda ser usada de nuevo por el sistema.

Si el recolector de basura no existiese, al cabo de muchas ejecuciónes de varios programas tendríamos la memoria llena de basura y de posiciones inservibles para el sistema, y esta tarea recaería en la pericia del programador para liberarla.

Ahora lo que debemos pensar es como hacemos que un objeto que hemos creado le indicamos que deje de apuntar a esa dirección de memoria para que el recolector de basura actue. Esto es bien sencillo, basta con usar la palabra reservada “null”, que índica que el objeto no apunta ya a nada. Por ejemplo:

```
calculadora c=new calculadora(); // Creo el objeto c
System.out.println("Método sumar(int v1,int v2):"+c.sumar(2,3));
c = null; // El objeto c deja de apuntar a memoria
```

Si intentasemos usar después del “null” el objeto calculadora obtendríamos un error.

Cuando el programa finaliza, aunque no pongamos los objetos a “null”, automaticamente al finalizar dejan de apuntar a las direcciones de memoria, con lo cual el recolector haría su tarea igualmente.

4.7 Ejercicios propuestos

1. Desarrolla una clase Cancion con los siguientes atributos:
   - **titulo**: una variable String que guarda el título de la canción.
   - **autor**: una variable String que guarda el autor de la canción.

los siguientes constructores:
   - **Cancion(String, String)**: constructor que recibe como parámetros el título y el autor de la canción (por este orden).
   - **Cancion()**: constructor predeterminado que inicializa el título y el autor a cadenas vacías.

y los siguientes métodos:
   - **dameTitulo():** devuelve el título de la canción.
   - **dameAutor():** devuelve el autor de la canción.
   - **ponTitulo(String):** establece el título de la canción.
   - **ponAutor(String):** establece el autor de la canción.

2. Escriba un programa para el que defina una clase llamada NOMBRE con los datos miembro de tipo String siguientes: Primer Nombre, Segundo Nombre, Primer Apellido, Segundo Apellido. Los datos miembros deben ser privados y los métodos/funciones deben ser públicos. La clase debe tener las siguientes funciones:
   - Constructor predeterminado que debe establecer todos los datos miembro en blanco.
   - Constructor que recibe como parámetros los datos.
   - Método imprimir1 para mostrar los datos en el siguiente formato:
     MARCO ANTONIO ZARZO MARTINEZ
   - Método imprimir2 para mostrar los datos en el siguiente formato:
     ZARZO MARTINEZ, MARCO ANTONIO

El programa principal debe crear dos nombres, y para cada uno de ellos, una vez metidos los datos se debe mostrar los nombres en los dos formatos diferentes.

3. Desarrolla una clase Cafetera con atributos:
   - **capacidadMaxima** (la cantidad máxima de café que puede contener la cafetera)
   - **cantidadActual** (la cantidad actual de café que hay en la cafetera).

Implementa, al menos, los siguientes constructores:
   - Constructor predeterminado: establece la capacidad máxima en 1000 (c.c.) y la actual en cero (cafetera vacía).
   - Constructor con la capacidad máxima de la cafetera; inicializa la cantidad actual de café igual a la capacidad máxima.
   - Constructor con la capacidad máxima y la cantidad actual. Si la cantidad actual es mayor que la capacidad máxima de la cafetera, la ajustará al máximo.

y los siguientes métodos:
   - **llenarCafetera():** pues eso, hace que la cantidad actual sea igual a la capacidad.
   - **servirTaza(int):** simula la acción de servir una taza con la capacidad indicada. Si la cantidad actual de café “no alcanza” para llenar la taza, se sirve lo que quede.
   - **vaciarCafetera():** pone la cantidad de café actual en cero.
   - **agregarCafe(int):** añade a la cafetera la cantidad de café indicada.

4. Crear una clase reloj que va a contener horas, minutos y segundos. Se debe crear un constructor por defecto que nos pone la hora a cero. Después vamos a tener otro constructor al cual le pasaremos las horas, minutos y segundos para que lo inicialice, la clase debe de tener métodos para cambiar las horas, minutos y segundos independientemente, además de un método para imprimir por pantalla la hora, así mismo también tendremos un método que se llama pasar_un_minuto que nos va mostrando por pantalla como pasan los segundos hasta que transcurre un minuto. Se debe comprobar que los valores introducidos son válidos. El formato del reloj será el siguiente 00:00:00. Ejemplo:
La hora actual es: 15:34:45
Si cambiamos la hora → la nueva hora es: 22:23:12
Si ponemos en marcha el reloj → Pasa un minuto...
22:23:12
22:23:13
22:23:14
. . .
22:23:59
22:24:00

5. Crea una clase Cuenta (bancaria) con atributos para el número de cuenta (un entero largo), el DNI del cliente (otro entero largo, no lleva letra), el saldo actual y el interés anual que se aplica a la cuenta (porcentaje). Cuando se crea una cuenta se genera un número aleatorio para el número de cuenta. Define en la clase los siguientes métodos:
- Constructor por defecto con los String vacíos y los números a cero.
- Constructor con DNI, saldo e interés.
- **actualizarSaldo():** actualizará el saldo de la cuenta aplicándole el interés diario (interés anual dividido entre 365 aplicado al saldo actual).
- **ingresar(double):** permitirá ingresar una cantidad en la cuenta.
- **retirar(double):** permitirá sacar una cantidad de la cuenta (si hay saldo).
- Método que nos permita mostrar todos los datos de la cuenta.

6. Crear una clase Empleado que contiene lo siguiente:
- **Nombre**: Nombre completo del empleado.
- **DNI**: Entero largo con el DNI del empleado sin letra.
- **Puesto**: Puesto de trabajo del empleado.
- **Sueldo base**: Decimal con el sueldo base del empleado.

Debemos crear los siguientes métodos:
- Constructor al que se le pasan como parámetros los datos del empleado.
- Método **mostrar_datos** que nos muestra los datos completos del empleado.
- Un método **set** para cada uno de los atributos del empleado.
- Un método **get** para el sueldo base.

7. Crear una clase Nomina, que va a incluir lo siguiente:
   - **Mes**: número de mes de la nomina.
   - **Anyo**: Año de la nomina.
   - **Retencion**: es un porcentaje (double) para calcular la nomina.
   - **Empleado**: Objeto empleado sobre el que se hace la nomina.

Vamos a tener el siguiente método:
   - **Calcular**: Nos calcula el sueldo limpio (al sueldo base se le quita la retención).

8. Cree una clase llamada Fecha, que incluya tres atributos de instancia:
   - un mes (tipo int)
   - un día (tipo int)
   - un año (tipo int).

Su clase debe tener un constructor que inicialice las tres variables de instancia, y debe asumir que los valores que se proporcionan son correctos.
Proporcione un método set un método get para cada variable de instancia. Proporcione un método mostrarFecha, que muestre el mes, día y año, separados por barras diagonales (/). Escriba una aplicación de prueba llamada PruebaFecha, que demuestre las capacidades de la clase Fecha.

9. Crear una clase Pajaro que tenga tres atributos uno será el tipo de pájaro que es un String, y otro será el color, que también es un String, y la edad de tipo int. Después crear Un constructor para inicializar los pájaros, y a continuación un método que se llama Piar que muestra por pantalla el tipo del pájaro seguido del texto “PIO”.
Después crear una clase Jaula la cual va a constar de un total de 3 pájaros. El constructor debe permitirnos pasarle los 3 pájaros. También debe tener un método llamado moverJaula, que provoca que los 3 pájaros píen y se muestre por pantalla.

10. Crear una clase Dado, que tiene un número entero como atributo. El constructor no recibe parámetros y lo que hace es inicializar el número entero a 1. Tendremos un método llamado lanzarDado que lo que hace es asignarle un valor aleatorio entre 1 y 6 al atributo del Dado, a parte mostrará por pantalla el resultado, y el mismo método nos devolverá el resultado.
Crear en un método main dos dados y hacer que se lancen continuamente hasta que en ambos salga un 6.

11. Diseñar una clase mcd con un método recursivo que permita calcular el máximo común divisor de dos números enteros y positivos por el algoritmo de Euclides. Para obtener el máximo común divisor de dos números enteros y positivos, a y b por el algoritmo de Euclides, debe dividirse a ente b y, si el resto es distinto de cero, repetir la división tras dar a a el valor de b y a b el resto. La condición de terminación o de salida será cuando el resto sea cero.

4.8 Ejercicios adicionales

1. *Crear una clase llamada punto. Un punto va a constar de una coordenada en X y otra en Y. Tendemos un constructor para iniciar las coordenadas y un método para cambiarlas, además de un método para devolver las coordenadas X y otro para las Y. Después crear una clase Linea que consta de 2 puntos. Debe tener un constructor que Permita pasarle 2 puntos para inicializarla. Después tendremos un método llamado calcularLongitud que nos permita calcular la longitud de la línea.*

2. Usándo la clase punto del ejercicio 1 hay que crear una clase triángulo que consta de 3 puntos y una clase rectángulo que consta de 4 puntos. Cada una de ellas debe tener dos constructores, uno para pasarle las coordenadas y otro para pasarle los puntos. Tanto el triángulo como el rectángulo van a tener un método que se llama calcularArea que nos va a calcular el área de esa figura en base a los puntos.

3. *Crear las siguientes clases:*

- Crear una clase Piloto que tiene como atributos el nombre, y la experiencia del piloto que es un valor decimal entre 0 y 1, debemos crear un constructor para inicializar los datos, también debe tener un método para cambiar la experiencia del piloto, otro para mostrar los datos y otro para devolver la experiencia del Piloto.

*- Después crearemos una clase Coche que tiene un modelo, una velocidad máxima con decimales, una distancia recorrida con decimales , y un Piloto. El Coche debe tener al menos un constructor que nos permita pasarle el modelo, la velocidad máxima, y el Piloto, la distancia se pondrá igual a cero. Debe tener un método que me permita cambiar el Piloto, otro llamado correr que incrementa la distancia recorrida en base a la siguiente fórmula: distancia=distancia+experienciaPiloto*velocidadMaxima y devuelve la distancia recorrida. Y por último un método que nos devuelve los datos del Coche.*

- Crear una clase Carrera que consta de 4 Coches conducidos por 4 Pilotos, y también consta de un atributo que es la distancia máxima de la carrera. Debe tener un constructor que nos permita inicializar la Carrera con los 4 Coches y la distancia máxima de la carrera. Por último tiene un método que se llama iniciarCarrera que cuando se ejecuta hace correr a los 4 coches hasta que uno de ellos alcanza la distancia máxima de la carrera y se proclama ganador.

Ejecutar al menos una carrera para comprobar el correcto funcionamiento.

4.9 Solución de ejercicios de Auto-Evaluación

Solución Auto-evaluación 4.1:

La clase Coche.java:

```
public class coche {
	// Aquí defino los atributos
	String modelo;
	int deposito, num_ruedas;
	boolean encendido;
	// Ahora defino su constructor, incialmente esta apagado
	public coche(String nuevoModelo, int dep, int ruedas) {
		modelo = nuevoModelo;
		deposito = dep;
		num_ruedas = ruedas;
		encendido = false;
	}
	// Ahora defino sus métodos
	public void encender() {
		if (deposito>0){
			if (encendido==false){
				encendido=true;
				System.out.println("Coche "+modelo+" encendido.");
			}else{
				System.out.println("El coche "+modelo+" ya está
encendido.");
			}
		}else{
			System.out.println(modelo+" no se puede encender.
Depósito vacío.");
		}
	}
	public void apagar() {
		if (encendido==true){
			encendido=false;
			System.out.println("Coche "+modelo+" apagado.");
		}else{
			System.out.println("El coche "+modelo+" ya está
apagado.");
		}
	}
	public void correr() {
		if (encendido==true){
			if (deposito>0){
				deposito--;
				System.out.println("El coche "+modelo+" corre.
Depósito:"+deposito);
			}else{
```

```
                        System.out.println("No se puede correr. Depósito
vacío.");
                        this.apagar();
                    }
                }else{
                    System.out.println("Enciende antes el coche
"+modelo+".");
                }
        }
}
```

El main:

```
public static void main(String[] args){
            coche c1=new coche("BMW",4,3);
            coche c2=new coche("AUDI",4,3);
            c1.correr();
            c1.apagar();
            c1.encender();
            c1.encender();
            c1.correr();
            c1.correr();
            c1.correr();
            c1.correr();
            c1.correr();
            c1.correr();
            c1.encender();
            c2.encender();
            c2.correr();
            c2.apagar();
            c2.correr();
            c2.apagar();
        }
```

Y el resultado sería:

```
Enciende antes el coche BMW.
El coche BMW ya está apagado.
Coche BMW encendido.
El coche BMW ya está encendido.
El coche BMW corre. Depósito:3
El coche BMW corre. Depósito:2
El coche BMW corre. Depósito:1
El coche BMW corre. Depósito:0
No se puede correr. Depósito vacío.
Coche BMW apagado.
Enciende antes el coche BMW.
BMW no se puede encender. Depósito vacío.
```

```
Coche AUDI encendido.
El coche AUDI corre. Depósito:3
Coche AUDI apagado.
Enciende antes el coche AUDI.
El coche AUDI ya está apagado.
```

Solución Auto-evaluación 4.2:

Este método es estático porque de lo contrario necesitaría antes que lo instanciasen para poder hacer una llamada. Es el método a partir del cual comenzamos la ejecución de nuestro programa, por tanto no podemos crear una instancia previa de él.

Solución Auto-evaluación 4.3:

```
import java.util.Scanner;

public class menuMain {
	static int menu(){
		int op;
		Scanner sc=new Scanner(System.in);
		System.out.println("\n MENU\n**********\n");
		System.out.println("1. OPCION 1");
		System.out.println("2. OPCION 2");
		System.out.println("0. SALIR");
		System.out.print("Elige una opción:");
		op=sc.nextInt();
		return op;
	}

	public static void main(String[] args) {
		int opcion;
		do{
			opcion=menu();
			if (opcion<0 || opcion>2){
				System.err.println("Opción incorrecta.");
			}
		}while(opcion!=0);
		System.out.println("Has salido del menú.");
	}
}
```

Solución Auto-evaluación 4.4:

La clase familia.java:

```
public class familia {
	String nombre;
	static String apellido,direccion;
	int edad;
```

```
	public familia(String nom, int ed){
		nombre=nom;
		edad=ed;
	}

	public void mostrar(){
		System.out.println("Nombre:"+nombre+"\tApellido:"
			+apellido+"\tEdad:"+edad+"\tDirección:"+direccion);
	}

	public static void setApellido(String ape){
		apellido=ape;
	}

	public static void setDireccion(String dir){
		direccion=dir;
	}
}
```

Y el main:

```
	public static void main(String[] args) {
		familia.setApellido("Terol");
		familia.setDireccion("C./Miguel Hernandez, 5");
		familia andres=new familia("Andrés",61);
		familia helios=new familia("Helios",27);
		familia david=new familia("David",26);
		familia carmen=new familia("Carmen",63);

		andres.mostrar();
		helios.mostrar();
		david.mostrar();
		carmen.mostrar();
	}
```

Y el resultado sería:

```
Nombre:Andrés     Apellido:Terol    Edad:61     Dirección:C./Miguel Hernandez, 5
Nombre:Helios     Apellido:Terol    Edad:27     Dirección:C./Miguel Hernandez, 5
Nombre:David      Apellido:Terol    Edad:26     Dirección:C./Miguel Hernandez, 5
Nombre:Carmen     Apellido:Terol    Edad:63     Dirección:C./Miguel Hernandez, 5
```

Solución Auto-evaluación 4.5:

Si lo resolvemos de manera iterativa (con bucles) la solución sería:

```
public class mainIterativo {
	public static int expIterativo(int base,int exp){
		int res=1;
		while(exp>0){
```

```
                res=res*base;
                exp--;
            }
            return res;
        }

        public static void main(String[] args) {
            System.out.println("El exponencial de 2 elevado a 3 es "
                +expIterativo(2,3));
        }
}
```

Ayudándonos de la solución vista arriba, si lo resolvemos de manera recursiva la solución sería:

```
public class mainRecursivo {
        public static int expRecursivo(int base,int exp){
            if (exp==0){
                return 1;
            }else{
                return base*expRecursivo(base,--exp);
            }
        }

        public static void main(String[] args) {
            System.out.println("El exponencial de 2 elevado a 3 es "
                +expRecursivo(2,3));
        }
}
```

Solución Auto-evaluación 4.6:

Si lo resolvemos de manera iterativa (con bucles) la solución sería:

```
public class mainIterativo {
        public static String reverseIterativa(String palabra){
            String inversa="";
            int pos=palabra.length()-1;
            do{
                inversa=inversa+palabra.charAt(pos);
                pos--;
            }while(pos>=0);
            return inversa;
        }

        public static void main(String[] args){
            System.out.println("La inversa de Hola es "
                +reverseIterativa("Hola"));
        }
}
```

Ayudándonos de la solución vista arriba, si lo resolvemos de manera recursiva la solución sería:

```
public class mainRecursivo {
	public static String reverseRecursiva(String palabra){
		if(palabra.length()==1){
			return palabra;
		} else {
			return palabra.charAt(palabra.length()-1)
		+reverseRecursiva(palabra.substring(0, palabra.length()-1));
		}
	}

	public static void main(String[] args){
		System.out.println("La inversa recursiva de Hola es "
			+reverseRecursiva("Hola"));
	}
}
```

Lo que hacemos es quitar la última letra, y mandamos a la llamada recursiva el resto, de tal manera que sería algo como esto:

reverseRecursiva(“Hola”)

“a”+reverseRecursiva(“Hol”)

“l”+reverseRecursiva(“Ho”)

“o”+reverseRecursiva(“H”)

y por último reverseRecursiva(“H”) nos dá como resultado “H” y ya podemos empezar a desapilar, dando como resultado “aloH”.