UNIDAD DE TRABAJO 5

CLASES Y OBJETOS

Introducción

Si nos paramos a observar el mundo que nos rodea, podemos apreciar que casi todo está formado por objetos. Existen coches, edificios, sillas, mesas, semáforos, ascensores e incluso personas o animales. Todos ellos pueden ser considerados objetos, con una serie de características y comportamientos. Por ejemplo, existen coches de diferentes marcas, colores, etc. y pueden acelerar, frenar, girar, etc., o las personas tenemos diferente color de pelo, ojos, altura y peso y podemos nacer, crecer, comer, dormir, etc.

Los programas son el resultado de la búsqueda y obtención de una solución para un problema del mundo real. Pero ¿en qué medida los programas están organizados de la misma manera que el problema que tratan de solucionar? La respuesta es que muchas veces los programas se ajustan más a los términos del sistema en el que se ejecutarán que a los del propio problema.

Si **redactamos los programas utilizando los mismos términos de nuestro mundo real**, es decir, utilizando objetos, y no los términos del sistema o computadora donde se vaya a ejecutar, conseguiremos que éstos sean más legibles y, por tanto, más fáciles de modificar.

Esto es precisamente **lo que pretende la Programación Orientada a Objetos (POO)**, en inglés OOP (Object Oriented Programming), establecer una serie de técnicas que permitan trasladar los problemas del mundo real a nuestro sistema informático.

Ahora que ya conocemos la sintaxis básica de Java, es el momento de comenzar a utilizar las características orientadas a objetos de este lenguaje, y estudiar los conceptos fundamentales de este modelo de programación.

Fundamentos de la POO

Dentro de las distintas formas de hacer las cosas en programación, distinguimos dos paradigmas fundamentales:

- **Programación Estructurada,** se crean **funciones y procedimientos** que definen las acciones a realizar, y que posteriormente forman los programas.
- **Programación Orientada a Objetos,** considera los programas en términos de **objetos** y toda gira alrededor de ellos.

Pero ¿en qué consisten realmente estos paradigmas? Veamos estos dos modelos de programación con más detenimiento.

Programación Estructurada

Inicialmente, se programaba aplicando las técnicas de programación tradicional, también conocidas como **Programación Estructurada**. El problema se descomponía en unidades más pequeñas hasta llegar a acciones o verbos muy simples y fáciles de codificar. Por ejemplo, en la resolución de una ecuación de primer grado, lo que hacemos es descomponer el problema en acciones más pequeñas o pasos diferenciados:

- Pedir valor de los coeficientes.
- Calcular el valor de la incógnita.
- Mostrar el resultado.

Si nos damos cuenta, esta serie de acciones o pasos diferenciados no son otra cosa que verbos; por ejemplo el verbo pedir, calcular, mostrar, etc.

Programación Orientada a Objetos

Sin embargo, la Programación Orientada a Objetos aplica de otra forma diferente la técnica de programación "divide y vencerás".

Este paradigma **lo que hace es descomponer, en lugar de acciones, en objetos**.

El principal **objetivo sigue siendo descomponer el problema en problemas más pequeños, que sean fáciles de manejar y mantener, fijándonos en cuál es el escenario del problema e intentando reflejarlo en nuestro programa**.

O sea, se trata de trasladar la visión del mundo real a nuestros programas. Por este motivo se dice que **la Programación Orientada a Objetos aborda los problemas de una forma más natural,** entendiendo como natural que está más en contacto con el mundo que nos rodea.

La Programación Orientada a Objetos es un sistema o conjunto de reglas que nos ayudan a descomponer la aplicación en objetos. A menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia directa entre el espacio del problema y el espacio de la solución. ¿Pero en la práctica esto qué quiere decir? Pues que a la hora de escribir un programa, nos fijaremos en los objetos involucrados, sus características comunes y las acciones que pueden realizar. Una vez localizados los objetos que intervienen en el problema real (espacio del problema), los tendremos que trasladar al programa informático (espacio de la solución). Con este planteamiento, la solución a un problema dado se convierte en una tarea sencilla y bien organizada.

VENTAJAS DE LA POO

Según lo que hemos visto hasta ahora, **un objeto es cualquier entidad que podemos ver o apreciar.**

El concepto fundamental de la Programación Orientada a Objetos es, precisamente, los objetos. Pero **¿qué beneficios aporta la utilización de objetos?** Fundamentalmente la posibilidad de representar el problema en términos del mundo real, que como hemos dicho están más cercanos a nuestra forma de pensar, pero existen otra serie de ventajas como las siguientes:

- **Comprensión**. Los conceptos del espacio del problema se hayan reflejados en el código del programa, por lo que la mera lectura del código nos describe la solución del problema en el mundo real.

- **Modularidad.** Facilita la modularidad del código, al estar las definiciones de objetos en módulos o archivos independientes, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.

- **Fácil mantenimiento.** Cualquier modificación en las acciones queda automáticamente reflejada en los datos, ya que ambos están estrechamente relacionados. Esto hace que el mantenimiento de las aplicaciones, así como su corrección y modificación sea mucho más fácil. Por ejemplo, podemos querer utilizar un algoritmo más rápido, sin tener que cambiar el programa principal. Por otra parte, al estar las aplicaciones mejor organizadas, es más fácil localizar cualquier elemento que se quiera modificar y/o corregir. Esto es importante ya que se estima que los mayores costes de software no están en el proceso de desarrollo en sí, sino en el mantenimiento posterior de ese software a lo largo de su vida útil.

- **Seguridad**. La probabilidad de cometer errores se ve reducida, ya que no podemos modificar los datos de un objeto directamente, sino que debemos hacerlo mediante las

acciones definidas para ese objeto. Imaginemos un objeto lavadora. Se compone de un motor, tambor, cables, tubos, etc. Para usar una lavadora no se nos ocurre abrirla y empezar a manipular esos elementos, ya que lo más probable es que se estropee. En lugar de eso utilizamos los programas de lavado establecidos. Pues algo parecido con los objetos, no podemos manipularlos internamente, sólo utilizar las acciones que para ellos hay definidas.

- **Reusabilidad**. Los objetos se definen como entidades reutilizables, es decir, que los programas que trabajan con las mismas estructuras de información pueden reutilizar las definiciones de objetos empleadas en otros programas, e incluso las acciones definidas sobre ellos. Por ejemplo, podemos crear la definición de un objeto de tipo persona para una aplicación de negocios y deseamos construir a continuación otra aplicación, digamos de educación, en donde utilizamos también personas, no es necesario crear de nuevo el objeto, sino que por medio de la reusabilidad podemos utilizar el tipo de objeto persona previamente definido.

PRINCIPIOS DE LA POO

Cuando hablamos de Programación Orientada a Objetos, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

- **Abstracción.** Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la **clase**.
  Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto.
  Por ejemplo, si pensamos en una clase **Vehículo** que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como **Coche** y **Camión**. Entonces se dice que **Vehículo** es una abstracción de **Coche** y de **Camión**.

- **Modularidad.** Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crea a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.

- **Encapsulación es** el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto **Persona** y otro **Coche. Persona** se comunica con el objeto **Coche** para llegar a su destino, utilizando para ello las acciones que Coche tenga definidas como por ejemplo conducir. Es decir, **Persona**

utiliza **Coche** pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.

- **Jerarquía**. Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía **"es un"** llamada **generalización** o **especialización** y la jerarquía **"es parte de"**, llamada **agregación**. Conviene detallar algunos aspectos:

    - La generalización o especialización, también conocida como **herencia**, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase **CochedeCarreras** a partir de la clase **Coche**, y así sólo tendremos que definir las nuevas características que tenga.

    - La agregación, también conocida como **inclusión**, permite agrupar objetos relacionados entre sí dentro de una clase. Así, un **Coche** está formado por **Motor**, **Ruedas**, **Frenos** y **Ventanas**. Se dice que **Coche** es una agregación y **Motor**, **Ruedas**, **Frenos** y **Ventanas** son agregados de **Coche**

- **Polimorfismo**. Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase **Animal** y la acción de expresarse. Nos encontramos que cada tipo de **Animal** puede hacerlo de manera distinta, los **Perros** ladran, los **Gatos** maullan, las **Personas** hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo **Animal**, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate.

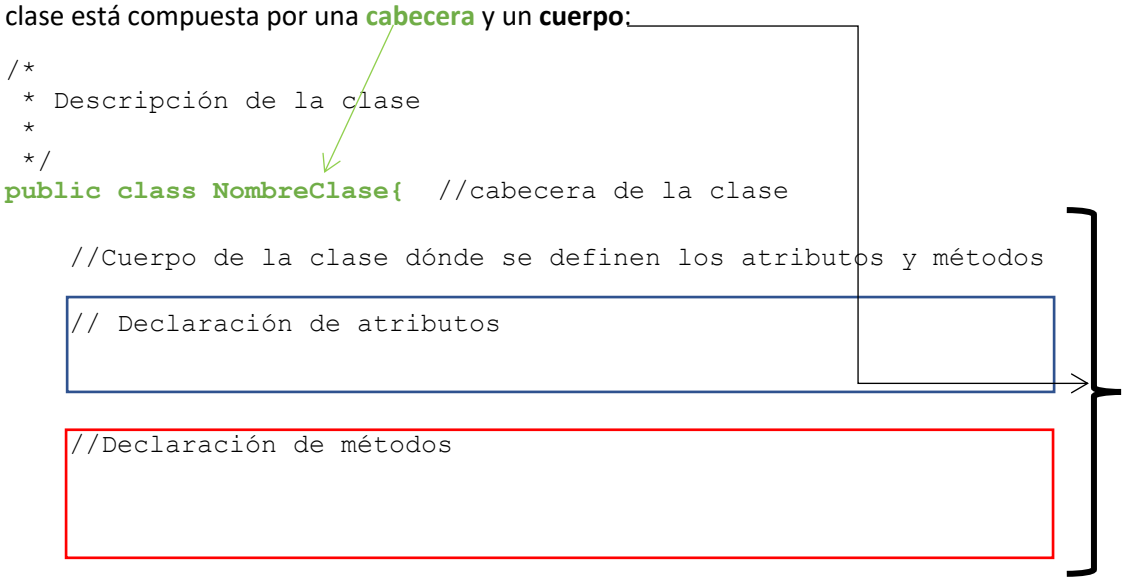
Clases

Las clases constan de datos y métodos que resumen las características comunes de un conjunto de objetos. Un programa informático está compuesto por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.

En otras palabras, **una clase es una plantilla o prototipo donde se especifican**:

- Los **atributos** comunes a todos los objetos de la clase. Son las características que definen a los objetos. Por ejemplo: en el caso de los vehículos podrían ser su marca, modelo, color, etc.
- Los **métodos** que pueden utilizarse para manejar esos objetos. Son las operaciones que se pueden hacer con los objetos. Por ejemplo: arrancar, frenar, cambiar de marcha, cambiar su color, etc.

Para declarar una clase en Java se utiliza la palabra reservada `class`. La declaración de una clase está compuesta por una **cabecera** y un **cuerpo**:



```
/*
 * Descripción de la clase
 *
 */
public class NombreClase{   //cabecera de la clase

    //Cuerpo de la clase dónde se definen los atributos y métodos

    // Declaración de atributos


    //Declaración de métodos


   //el método main indica que se trata de una clase principal
    //a partir de la cual se va a empezar a ejecutar el programa
    public static void main(String[] argumentos){


    }//fin main

}//fin clase
```

CABECERA

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

```
[modificadores] class <NombreClase> [extends <NombreSuperClase>][implements <NombreInter-
face1>]
[[implements<NombreInterface2>] …] {
```

En **[modificadores]** puede aparecer:

- Modificador **public**. Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase. Es decir, desde cualquier otra parte del programa. Si no se especifica este modificador, la clase sólo podrá ser utilizada desde clases que estén en el mismo **paquete**. El concepto de paquete lo veremos 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 es posible crear objetos de esa clase (habrá que utilizar clases que hereden de ella). En este momento es posible que te parezca que no tenga sentido que esto pueda suceder puedes crear objetos de esa clase, ¿para qué la quieres?), pero puede resultar útil a la hora de crear una jerarquía de clases. Esto lo verás también más adelante al estudiar el concepto de **herencia**.

- Modificador **final**. Indica que no podrás crear clases que hereden de ella. También volverás a este modificador cuando estudies el concepto de **herencia**. Los modificadores **final** y **abstract** son excluyentes (sólo se puede utilizar uno de ellos).

En **[extends]** se indica para decir que esa clase es subclase de otra. Es decir, que hereda todos sus atributos y métodos. La herencia se estudiará en la siguiente unidad de trabajo

En **[implements]** indica que está implementando una interface.

Nota: poco a poco iremos viendo con ejercicios y ejemplos todos estos conceptos.

A la hora de implementar una clase Java debes tener en cuenta:

- Por convenio, se ha decidido que en lenguaje Java los nombres de las clases deben de **empezar por una letra mayúscula**. Así, cada vez que observes en el código una palabra con la primera letra en mayúscula sabrás que se trata de una clase sin necesidad de tener que buscar su declaración. Además, **si el nombre de la clase está formado por varias palabras, cada una de ellas también tendrá su primera letra en mayúscula**. Siguiendo esta recomendación, algunos ejemplos de nombres de clases podrían ser: Recta, Circulo, Coche, CocheDeportivo, Jugador, JugadorFutbol, AnimalMarino, AnimalAcuatico, etc.

- El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo (**clase principal 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 (por ejemplo en C++, serían sendos archivos con extensiones ".h" y ".cpp").

Atributos

Toda clase tiene una **serie de atributos que describen las propiedades o características de los objetos** de la clase.

Un atributo viene dado por:

- un nombre identificador
- un tipo de dato

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**).

Como ya verás más adelante, al estudiar el concepto de **encapsulación**, **lo normal 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.**

Métodos

Toda clase tiene una **serie de métodos que son las operaciones disponibles para manipular a los objetos de la clase.**

Los métodos son ejecutados por los objetos de la clase cuando reciben un mensaje

Los métodos pueden devolver o no algún valor (funciones o procedimientos) y pueden tener o no parámetros.

De acuerdo con **el tipo de acción**, podemos hacer la siguiente **clasificación**:

- **Métodos de construcción:** construyen objetos normalmente dando valor a sus atributos.
- **Métodos de destrucción**: destruyen objetos liberando la memoria que ocupan.
- **Métodos de modificación**: modifican el estado del objeto, es decir, cambian el valor a algún atributo.
- **Métodos de acceso o selección**: comprueban el estado del objeto (valor de algún atributo).

De acuerdo con la **accesibilidad**, podemos clasificarlos en:

- **Métodos públicos**: aquellos que son accesibles desde fuera.
- **Métodos internos o privados**: aquellos que sólo son accesibles desde dentro de la propia clase.

Un método puede comportarse de formas distintas dependiendo del objeto que le llame (principio de polimorfismo)

Ejemplo

Crea una clase Persona con los siguientes atributos: nombre, apellido1, apellido2, dni, profesión, edad. Los atributos deben ser privados y declarados del tipo que consideres más conveniente para cumplir la función que se intuye por su nombre. También debe tener métodos para poder asignar valor y obtenerlo de sus atributos, así como un constructor que reciba por parámetros los datos de los atributos.

```
/*
 * Ejemplo de cómo crear una clase
 * Crea una clase Persona con los siguientes atributos:
 * nombre, apellido1, apellido2, dni, profesión, edad.
 * También debe tener métodos para poder asignar valor y obtenerlo de
sus atributos, así como un constructor que reciba por parámetros los
datos de los atributos.
 */
public class Persona {
	// Atributos de la clase Persona, privados (ppio encapsulación)
	private String nombre;
	private String apellido1;
	private String apellido2;
	private String dni;
	private String profesion;
	private int edad;



	public Persona(String nombre, String apellido1,
```

```
                  String apellido2, String dni,
                  String profesion, int edad) {
            // método constructor, que asigna valores que recibe como
parámeetros a los
            // atributos
            this.nombre = nombre;
            this.apellido1 = apellido1;
            this.apellido2 = apellido2;
            this.dni = dni;
            this.profesion = profesion;
            this.edad = edad;
      }

      public String getNombre() {
            // devuelve el nombre
            return nombre;
      }

      public void setNombre(String nombre) {
            // asigna el nombre
            this.nombre = nombre;
      }

      public String getApellido1() {
            // devuelve el apellido
            return apellido1;
      }

      public void setApellido1(String apellido1) {
            // asigna apellido
            this.apellido1 = apellido1;
      }

      public String getApellido2() {
            return apellido2;
      }

      public void setApellido2(String apellido2) {
            this.apellido2 = apellido2;
      }

      public String getDni() {
            return dni;
      }

      public void setDni(String dni) {
            this.dni = dni;
      }

      public String getProfesion() {
            return profesion;
      }

      public void setProfesion(String profesion) {
            this.profesion = profesion;
      }

      public int getEdad() {
            return edad;
      }
```

```
	public void setEdad(int edad) {
		this.edad = edad;
	}
}
```

Observaciones:

- El nombre de la clase debe coincidir con el nombre del fichero
- Los atributos deben ser privados para que no sean accesibles desde fuera de la clase. Recordemos el principio de encapsulación.
- Por el motivo anterior, por cada uno de los atributos que definamos debemos tener dos métodos: get/set
- El método constructor, inicializa con los valores el objeto. Puede haber más de uno.

Objetos

Concepto

DEFINICIÓN 1

Un objeto es una estructura compleja que consta de:

- Una serie de ***datos***, que son los valores concretos que tiene el objeto respecto de una serie de propiedades denominadas atributos.
- Una serie de ***operaciones*** que permiten manipular los datos del objeto. A estas operaciones las llamamos métodos y el objeto será capaz de ejecutarlos cuando reciba un mensaje.

El hecho de que un objeto englobe los datos y las operaciones para manejarlos se conoce como ENCAPSULACIÓN

DEFINICIÓN 2

Un objeto es una entidad que exhibe 3 características:

- **Identidad**. Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.

- **Estado (atributos)**. El estado de un objeto viene determinado por una serie de parámetros o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto Coche, el estado estaría definido por atributos como `Marca`, `Modelo`, `Color`, `Cilindrada`, etc.

- **Comportamiento (métodos)**. Son las acciones que se pueden realizar sobre el objeto. En otras palabras, son los métodos o procedimientos que realiza el objeto. Siguiendo con el ejemplo del objeto **Coche**, el comportamiento serían acciones como: `arrancar()`, `parar()`, `acelerar()`, `frenar()`, etc.

Atributos y métodos

Como acabamos de ver todo objeto tiene un estado y un comportamiento. Concretando un poco más, las partes de un objeto son:

- **Campos, Atributos o Propiedades:** Parte del objeto que almacena los datos. También se les denomina V**ariables** M**iembro**. Estos datos pueden ser de cualquier tipo primitivo (`boolean`, `char`, `int`, `double`, etc) o ser su vez ser otro objeto. Por ejemplo, un objeto de la clase Coche puede tener un objeto de la clase Ruedas.
- **Métodos o Funciones Miembro:** Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La idea principal es que **el objeto reúne en una sola entidad los datos y las operaciones, y para acceder a los datos privados del objeto debemos utilizar los métodos que hay definidos para ese objeto.**

La **única forma de manipular la información del objeto es a través de sus métodos.** Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera inadecuada. **Se dice que los datos y los métodos están encapsulados dentro del objeto.**

Interacción entre objetos

Dentro de un programa los objetos se comunican llamando unos a otros a sus métodos. Los métodos están dentro de los objetos y describen el comportamiento de un objeto cuando recibe una llamada a uno de sus métodos. En otras palabras, cuando un objeto, objeto1**,** quiere actuar sobre otro, objeto2**,** tiene que ejecutar uno de sus métodos. Entonces se dice que el objeto2 recibe un mensaje del objeto1.

Un **mensaje** es la acción que realiza un objeto. Un **método** es la función o procedimiento al que se llama para actuar sobre un objeto.

Los distintos mensajes que puede recibir un objeto o a los que puede responder reciben el nombre de **protocolo** de ese objeto.

El proceso de interacción entre objetos se suele resumir diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto determina "qué hacer con el mensaje" (ejecuta el código del método). Cuando se ejecuta un programa se producen las siguientes acciones:

- Creación de los objetos a medida que se necesitan.
- Comunicación entre los objetos mediante el envío de mensajes unos a otros, o el usuario a los objetos.
- Eliminación de los objetos cuando no son necesarios para dejar espacio libre en la memoria del computador.

Uso de objetos

Una vez que hemos creado una clase, podemos crear objetos en nuestro programa a partir de esas clases.

Cuando creamos un objeto a partir de una clase se dice que hemos creado una **"instancia de la clase".** A efectos prácticos, "objeto" e "instancia de clase" son términos similares. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son de una clase particular.

Los objetos se crean a partir de las clases, y representan casos individuales de éstas.

Cualquier objeto instanciado de una clase (operador new) contiene una copia de todos los atributos definidos en la clase. En otras palabras, lo que estamos haciendo es reservar un espacio en la memoria del ordenador para guardar sus atributos y métodos. Por tanto, cada objeto tiene una zona de almacenamiento propia donde se guarda toda su información, que será distinta a la de cualquier otro objeto. A las variables miembro instanciadas también se les llama variables instancia. De igual forma, a los métodos que manipulan esas variables se les llama métodos instancia.

Declaración de objetos

Para la creación de un objeto hay que seguir los siguientes pasos:

- **Declaración**: Definir el tipo de objeto.
- **Instanciación**: Creación del objeto utilizando el operador new**.**

Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

```
<tipo> nombre_objeto;
```

Donde:

- **tipo** es la clase a partir de la cual se va a crear el objeto, y
- **nombre_objeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.

En este momento, como el objeto no ha sido instanciado tiene el valor null

Instanciación del objeto

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden `new` con la siguiente sintaxis:

```
nombre_objeto = new <Constructor_de_la_Clase>([<par1>, <par2>, ..., <parN>]);
```

Donde:

- **`nombre_objeto`** es el nombre de la variable referencia con la cual nos referiremos al objeto,
- **`new`** es el operador para crear el objeto,
- **`Constructor_de_la_Clase`** es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos, y
- **`par1-parN`**, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto.

Manipulación

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del **operador punto (.)** y el nombre del atributo o método que queremos utilizar. Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto.

Por ejemplo, para acceder a las variables instancia o atributos se utiliza la siguiente sintaxis:

```
nombre_objeto.atributo
```

¡¡OJO NO ACCEDER DE FORMA DIRECTA USAR MÉTODOS!!

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

```
nombre_objeto.método( [par1, par2, ..., parN] )
```

En la sentencia anterior `par1`, `par2`, etc. son los parámetros que utiliza el método. Aparece entre corchetes para indicar son opcionales.

Eliminación de objetos.

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados por el programa. A esta acción se le denomina **destrucción del objeto.**

En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador new. Esto evita que los programadores tengan que preocuparse de realizar la liberación de memoria.

El recolector de basura se ejecuta en modo segundo plano y de manera muy eficiente para no afectar a la velocidad del programa que se está ejecutando. Lo que hace es que periódicamente va buscando objetos que ya no son referenciados, y cuando encuentra alguno lo marca para ser eliminado. Después los elimina en el momento que considera oportuno.

Justo antes de que un objeto sea eliminado por el recolector de basura, se ejecuta su método `finalize()`. Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método `runFinalization()` de la clase `System`. La clase `System` forma parte de la Biblioteca de Clases de Java y contiene diversas clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

```
System.runFinalization();
```

Ejemplo

En el apartado anterior se modelo la clase Persona, pero no se llegó a utilizar. La clase persona es el tipo de dato. En este apartado se creará un programa Java que cree dos objetos Persona, con valores concretos.

```
public class ProgramaPersonas {
    public static void main(String[] argumentos) {

        // Declara e instancia objeto 1 persona
        Persona persona1 = new Persona("Juan",
                "Pérez", "Vidal",
                "1234567A", "Camarero", 28);
        // Uso del objeto
        System.out.print(persona1.getNombre() + " " +
persona1.getApellido1());
        System.out.print(" es " + persona1.getProfesion() + " y tiene
");
        System.out.println(persona1.getEdad() + " años de edad.");
        persona1.setProfesion("Programador");
        // Declara e instancia objeto 1 persona
        Persona persona2 = new Persona("Luis", "Martín", "Ruiz",
                "9876543S", "Médico", 58);
        System.out.print(persona2.getNombre() + " " +
persona2.getApellido1());
        System.out.print(" es " + persona2.getProfesion() + " y tiene
");
        System.out.println(persona2.getEdad() + " años de edad.");

        System.runFinalization();// Libera espacio
    }
}
```

Los atributos

Declaración de atributos

[modificadores] <tipo><nombreAtributo>;

- **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, puertaIzquierda, cuartoTrasero, equipoVecendor, sumaTotal, nombreCandidatoFinal, etc. Cualquier identificador válido de Java será admitido como nombre de atributo válido, pero es importante seguir este convenio para facilitar la legibilidad del código (todos los programadores de Java lo utilizan).

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. Es más que probable que no necesites utilizarlos en este módulo.

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 permitirá el acceso a este atributo desde todas las clases que estén dentro del mismo paquete (package) que esta clase (la que contiene el atributo que se está declarando). No es necesario escribir ninguna palabra reservada. Si no se pone nadase supone se desea indicar este modo de acceso.
- Modificador de acceso **public**. Indica que cualquier clase (por muy ajena o lejana que sea) tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (public).
- Modificador de acceso **private**. Indica que sólo 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 atributo. Es lo opuesto a lo que permite public.
- Modificador de acceso **protected**. En este caso se permitirá acceder al atributo desde cualquier subclase (lo verás más adelante al estudiar la herencia) de la clase en la que se encuentre declarado el atributo, y también desde las clases del mismo paquete.

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

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 para todos los objetos de una misma clase. Es decir, todas las clases compartirán ese mismo atributo con el mismo valor. Es un caso de miembro estático o miembro de clase: un **atributo estático** o **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**.

Atributos estáticos

Como ya has visto, el modificador **static** hace que el atributo sea común (el mismo) para todos los objetos de una misma clase. En este caso sí 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 el mismo para todos los objetos y tendrá un valor único independientemente de cada objeto. Es más, aunque no exista ningún objeto de tener 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

```
public class PersonaStatic {
    // Atributos de la clase Persona, privados (ppio encapsulación)
    private String nombre;
    private String apellido1;
    private String apellido2;
    private String dni;
    private String profesion;
    private int edad;
    static int contadorPersonas;

    public PersonaStatic(String nombre, String apellido1,
            String apellido2, String dni,
            String profesion, int edad) {
        // método constructor, que asigna valores que recibe como
parámeetros a los
        // atributos
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
        this.dni = dni;
        this.profesion = profesion;
        this.edad = edad;
    }

    public String getNombre() {
        // devuelve el nombre
        return nombre;
    }
```

```
public void setNombre(String nombre) {
    // asigna el nombre
    this.nombre = nombre;
}

public String getApellido1() {
    // devuelve el apellido
    return apellido1;
}

public void setApellido1(String apellido1) {
    // asigna apellido
    this.apellido1 = apellido1;
}

public String getApellido2() {
    return apellido2;
}

public void setApellido2(String apellido2) {
    this.apellido2 = apellido2;
}

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}

public String getProfesion() {
    return profesion;
}

public void setProfesion(String profesion) {
    this.profesion = profesion;
}
```

```
    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public static void incrementarContador() {
        contadorPersonas++;
    }

    public static int getContadorPersonas() {
            return contadorPersonas;
    }

    public static void setContadorPersonas(int contadorPersonas)
{
        PersonaStatic.contadorPersonas = contadorPersonas;
    }

}
```

El programa que usa la clase anterior:

```
public class ProgramaPersonasStatic {
    public static void main(String[] argumentos) {

        // Declara e instancia objeto 1 persona
        PersonaStatic persona1 = new PersonaStatic("Juan",
                "Pérez", "Vidal",
                "1234567A", "Camarero", 28);
        PersonaStatic.incrementarContador();

        PersonaStatic persona2 = new PersonaStatic("Mario",
        "Pérez", "Ruiz",
        "1111167A", "Jubilado", 68);

        PersonaStatic.incrementarContador();

        System.out.println("Se han creado: " +
PersonaStatic.getContadorPersonas() + " personas");


    }
}
```

Los métodos

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado.

Los métodos están compuestos por una **cabecera** y un **cuerpo**.

```
public tipo_dato_devuelto nombre_método (par1, par2,…parN)
{
    //declaración variables locales al método

    //instrucciones del método
}
```

La cabecera también tiene modificadores, en este caso hemos utilizado `public` para indicar que el método es público, lo cual quiere decir que le pueden enviar mensajes no sólo los métodos del objeto sino los métodos de cualquier otro objeto externo.

Parámetros

En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

- **Por valor.** El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.
- **Por referencia.** La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por tanto cualquier cambio en el dato seguirá modificado una vez que salgamos del método.

En el lenguaje Java, todas las variables se pasan por valor, excepto los objetos que se pasan por referencia. En Java, la declaración de un método tiene dos restricciones:

- **Un método siempre tiene que devolver un valor** (no hay valor por defecto). Este **valor de retorno** es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo `void`, que indica que el método no devuelve ningún valor.
- **Un método tiene un número fijo de argumentos.** Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llame, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de **parámetros** cuando aparecen en la declaración del método.

Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

```
public tipo_de_dato_devuelto nombre_metodo (lista_de_parametros);
```

Como vemos, el tipo de dato devuelto aparece después del modificador `public` y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

```
(tipo_parámetro1 nombre_parámetro1, ..., tipo_parámetroN nombre_parámetroN)
```

Cuando se llame al método, se deberá utilizar el nombre del método, seguido de los argumentos que deben coincidir con la lista de parámetros

Constructores

Un constructor es un método especial con el mismo nombre de la clase y que no devuelve ningún valor tras su ejecución.

La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

- **El constructor es invocado automáticamente en la creación de un objeto,** y sólo esa vez.
- **Los constructores no empiezan con minúscula**, como el resto de los métodos, ya que se llaman igual que la clase y las clases empiezan con letra mayúscula.
- **Puede haber varios constructores** para una clase.
- Como cualquier método, el constructor puede tener **parámetros** para definir qué valores dar a los atributos del objeto.
- El **constructor por defecto** es aquél que no tiene argumentos o parámetros. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos utilizando el constructor por defecto.
- **Es necesario que toda clase tenga al menos un constructor**. Si no definimos constructores para una clase, y sólo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: 0 para los tipos numéricos, false para los `boolean` y `null` para el tipo carácter y las referencias. Dicho constructor lo que hace es llamar al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.

Operador this

Los constructores y métodos de un objeto suelen utilizar el operador `this`. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos `this.nombre_atributo`, y así no habrá duda de a qué elemento nos estamos refiriendo.

EJEMPLO

Modela en Java una clase Pájaro con los siguientes métodos:

- **`pajaro()`**. Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.
- **`pajaro(String nombre, int posX, int posY)`**. Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.
- **`volar(int posX, int posY)`**. Método que recibe como argumentos dos enteros: `posX` y `posY`, y devuelve un valor de tipo `double` como resultado, usando la palabra clave `return`. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:

$$desplazamiento = \sqrt{posX * pos\ X + posY * posY}$$

Diseña un programa que utilice la clase Pájaro, cree una instancia de dicha clase y ejecute sus métodos.

Solución:

```
/* Fichero Pajaro.java*/
public class Pajaro {
    private String nombre;
    private int posX, posY;

    public Pajaro(String nombre, int posX, int posY) {
        this.nombre = nombre;
        this.posX = posX;
        this.posY = posY;
    }

    double volar(int posX, int posY) {
        double desplazamiento = Math.sqrt(posX * posX + posY * posY);
        this.posX = posX;
        this.posY = posY;
        return desplazamiento;
    }

}

/* Fichero ProgramaPajaro.java */
public class ProgramaPajaro {
    public static void main(String[] argumentos) {
        Pajaro loro = new Pajaro("Lu", 50, 50);
        double d = loro.volar(50, 50);
        System.out.println("El desplazamiento es: " + d);
    }
}
```

Métodos estáticos

Cuando trabajábamos con cadenas de caracteres utilizando la clase `String`, veíamos las operaciones que podíamos hacer con ellas: obtener longitud, comparar dos cadenas de caracteres, cambiar a mayúsculas o minúsculas, etc. Pues bien, sin saberlo estábamos utilizando métodos estáticos definidos por Java para la clase `String`. Pero ¿qué son los métodos estáticos? Veámoslo.

Los **métodos estáticos** son aquellos métodos definidos para directamente, sin necesidad de crear un objeto de dicha clase. También se llaman **métodos de clase**

Ya se expuso un ejemplo en el apartado de atributos estáticos.

Los paquetes

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

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

Uso de 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

Ejemplo: java.lang.String

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un **import** de cada una de ellas, 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.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, tan solo las que haya en el paquete

Incluir 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;

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

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

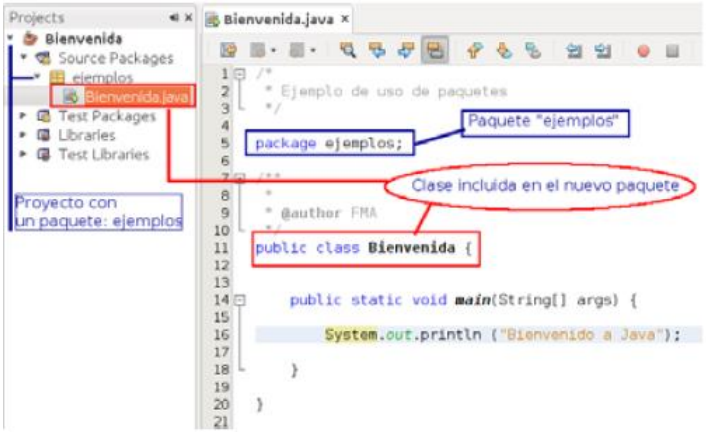
PASOS PARA CREAR UN PAQUETE

1. **Poner un nombre al 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) del archivo .java mediante el uso de la sentencia **package** tal y como has visto en el apartado anterior.

Este proceso ya lo has debido de llevar a cabo en unidades anteriores al compilar y ejecutar clases con paquetes. Estos pasos simplemente son para que te sirvan como recordatorio del procedimiento que debes seguir a a la hora de clasificar, jerarquizar y utilizar tus propias clases.



Librerías Java

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**.**