# Tarea online

**Título de la tarea:** Desarrollo de clases.

**Unidad:** 05

**Ciclo formativo y módulo:** DAM/DAW, Programación.

**Curso académico:** 2025/26

## ¿Qué contenidos o resultados de aprendizaje trabajaremos?

En esta tarea trabajaremos con el siguiente resultado de aprendizaje:

- **RA4.** *Desarrolla programas organizados en clases analizando y aplicando los principios de la programación orientada a objetos*.

# 1.- Descripción de la tarea

## Caso práctico

ajcespedes (Pixabay Lincense)

Una empresa que gestiona dispositivos con la capacidad de volar ha encargado a **BK Programación** la implantación de algún mecanismo que permita llevar un control de los dispositivos disponibles, así como gestionar los desplazamientos que realicen.

Para ello se está desarrollando un prototipo de clase que represente cada uno de esos dispositivos y que guarde información relacionada con su estado (en que posición se encuentran y a qué altura sobre el suelo están).

**Ada** ha repartido el trabajo entre los distintos componentes de la empresa, y a **Juan** le ha tocado diseñar e implementar el código correspondiente para representar el prototipo de clase.

**Juan** está dándole vueltas a la idea pensando en los atributos y métodos que serán necesarios para modelar el dispositivo:

— Habría que incluir un atributo que indique el **modelo** del dispositivo. Ese atributo sería inmutable a lo largo de la vida de todo el objeto. Sin embargo otros atributos, como la **altitud** a la que se encuentre, estarán más relacionados con el estado del objeto y podrán ir cambiando a lo largo de la vida de este, ¿no crees?

— Sí, parece bastante razonable. Además, también vendría bien disponer de atributos que fueran independientes de los objetos, y que dependieran directamente de la clase, incluso aunque no existieran aún objetos instanciados. — le contesta **María**, que está echándole una mano.

— ¡Claro! Imagina que queremos registrar en alguna parte **cuántos dispositivos se han creado** hasta el momento. ¿Qué mejor forma de representarlo que utilizando **atributos estáticos** para ello? ¡Buena idea! — le interpela **Juan**, mientras sigue dibujando un pequeño esquema de cómo podría quedar la clase que representaría a los dispositivos.

# ¿Qué te pedimos que hagas?

La tarea consistirá en la implementación de una **clase que nos permita gestionar una serie de dispositivos voladores (drones, helicópteros radiocontrol...)** (clase `DispositivoVolador`). Los objetos de esa clase dispondrán de unas **características inmutables** como, por ejemplo, su **modelo** o el **tipo de motor que posee** (eléctrico, combustión o híbrido). Además, estos objetos de tipo `DispositivoVolador` contendrán también atributos variables con **información de estado** que sí podrán ir cambiando como, por ejemplo, la **altitud a la que está el dispositivo** o su **posición en el eje X e Y** sobre el plano horizontal en ese momento.

Nordseher (Pixabay Lincense)

Por otro lado, se pretende también que la propia clase, independientemente de la existencia o no de instancias suyas, contenga **información global** como, por ejemplo, el **número de dispositivos** que hay creados.

A partir de la información contenida en los atributos podremos implementar **métodos** que definirán el **comportamiento de los objetos de la clase** ante determinadas acciones (desplazar, calcular el consumo, etc.).

# 1.1.- Atributos de la clase

Implementa en Java la clase `DispositivoVolador` declarando las propiedades o atributos necesarios para poder gestionar una serie de dispositivos que tienen la capacidad de volar y las acciones que pueden realizar.

ajcespedes (Pixabay Lincense)

Algunos atributos tendrán un valor que se definirá al instanciarse el objeto y ya nunca modificarán su contenido durante toda la vida del objeto (constantes). Podría decirse que definen algo así como la "naturaleza inmutable" o **características intrínsecas del objeto**:

- **modelo** del dispositivo volador;
- **tipo de motor** del dispositivo volador (`Electrico`, `Combustion` o `Hibrido`).

> *No olvides que los **nombres de todos los identificadores** (clase, métodos, parámetros, atributos, variables, constantes, etc.) deben cumplir las **convenciones establecidas para el lenguaje Java** (uso de mayúsculas, minúsculas, guiones, etc. dependiendo de la función de ese identificador: variable, constante, método, clase, etc.). Así mismo, recuerda también que para el caso de **atributos inmutables (constantes) privados** nos tomamos la licencia de usar la notación "Lower Camel Case" en lugar de escribirlo todo en mayúsculas.*

Otros atributos sí irán cambiando a lo largo de la vida del objeto y de alguna manera representarán el **estado del objeto** en cada instante. Por un lado, estarán los atributos referentes al estado del dispositivo volador:

- la **altitud actual** a la que está el dispositivo (en metros);
- la **posición** en la que se encuentra el dispositivo en el eje **X** (en puntos de coordenadas);
- la **posición** en la que se encuentra el dispositivo en el eje **Y** (en puntos de coordenadas).

> *A la hora de elegir los **tipos** más apropiados para representar toda esa información es conveniente que reflexiones sobre la utilidad de cada dato y los tipos disponibles en Java para poder representarlos. Por otro lado, no estaría mal que observaras los tipos de los parámetros en constructores y métodos, así como de los valores devueltos, pues también te podrán dar una pista importante de lo que se va a proporcionar como entrada al objeto y lo que se va a esperar de salida cuando nos comuniquemos con ellos.*

Además, también se dispondrá de un atributo que pertenecerá a la clase (atributo de clase), más que a un objeto en particular, compartiendo su valor para todos los objetos. Este atributo tendrá sentido siempre, independientemente de que existan o no instancias de la clase (**atributos estáticos**) y se accederá a él a través del nombre de la clase (y no de una referencia a un objeto):

- el **número de dispositivos** creados hasta ese momento (equivalente al número de objetos `DispositivoVolador` instanciados).

Por último, junto con ese atributo estático variable, la **clase** también va a contener una serie de **constantes** que serán de utilidad a la hora de realizar comprobaciones o asignaciones por omisión. Estas constantes serán **públicas** y tendrá acceso a ellas cualquier programador o programadora que utilice la clase para así poder asegurarse en sus programas de que no se superan los umbrales permitidos:

1. **altitud máxima** permitida para los dispositivos, en metros (**50.0**). Atributo `ALTITUD_MAXIMA`;
2. **rango de operación** del dispositivo, que es la distancia máxima que tiene de cobertura en puntos de coordenadas (**100**). Atributo `RANGO_OPERACION`;
3. **modelo por defecto** del dispositivo (**Dispositivo Generico**). Atributo `MODELO_DEFECTO`;
4. **tipo de motor por defecto** del dispositivo (**Electrico**). Atributo `TIPO_MOTOR_DEFECTO.`

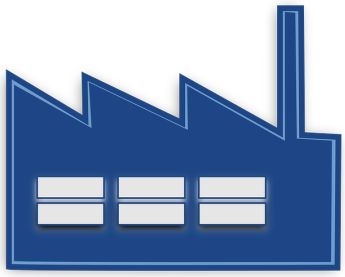
Se te proporciona explícitamente el **nombre de estos últimos atributos** porque van a ser **públicos** y ya hay programas desarrollados por otras personas que están usando esos valores (son valores de "*interfaz*" y, por tanto, públicos). En consecuencia, hemos tenido que ponernos de acuerdo con el resto de programadores del equipo de desarrollo en los nombres de todos aquellos elementos que sean públicos (nombre de la clase, nombre de los atributos públicos, nombres de los métodos públicos, orden y tipo de los parámetros en los métodos públicos, etc.). Para el resto de elementos de la clase (miembros privados, variables locales, etc.) se podrán utilizar los identificadores que consideres más oportunos, pues estarán ocultos a otros programadores y solo serán accesibles desde el código de los métodos de la propia clase.

Toda la información descrita en este apartado (**y solamente esta**) es la que se debe representar internamente la clase `DispositivoVolador`. No se necesitan más atributos. La valoración y decisión de los atributos necesarios para representar nuestros objetos de tipo `DispositivoVolador` ya ha sido tomada por los diseñadores de la aplicación y nosotros, como programadores finales, únicamente debemos limitarnos a implementar en Java las especificaciones detalladas que se nos acaban de proporcionar en este enunciado.

> *Recuerda que **los nombres de los identificadores (atributos) deben ser representativos de la información que están almacenando o van a gestionar**, tanto si son públicos como si no lo son. Procura elegir nombres apropiados.*

# 1.2.- Constructores y métodos "fábrica"

Tendrás que implementar dos **constructores** para la clase `DispositivoVolador`:



Clker-Free-Vector-Images (Pixabay License)

1. *Un **constructor con dos parámetros**: **modelo y tipo de motor**.*

Este constructor creará un nuevo objeto instancia de la clase con la información que se le proporciona a través de sus parámetros modelo y tipo de motor.

**Si alguno de los parámetros del constructor es incorrecto** o se produce alguna **circunstancia que no permita que se instancie el objeto**, deberá **lanzarse una excepción** y se detendrá el proceso de instanciación del objeto. En este caso, las posibles excepciones que podrían lanzarse son:

- si el constructor recibe un `null` en modelo, la excepción será de tipo `NullPointerException` y el mensaje de error de la excepción debería ser del tipo **"El modelo del dispositivo no puede ser nulo."**;
- si el constructor recibe una cadena vacía en modelo, la excepción será de tipo `IllegalArgumenException` y el mensaje de error de la excepción debería ser del tipo **"El modelo del dispositivo no puede estar vacío."**;
- si el constructor recibe un `null` en tipo de motor, la excepción será de tipo `NullPointerException` y el mensaje de error de la excepción debería ser del tipo **"El tipo de motor no puede ser nulo."**;
- si el constructor recibe una cadena vacía en el tipo de motor, la excepción será de tipo `IllegalArgumenException` y el mensaje de error de la excepción debería ser del tipo **"El tipo de motor no puede estar vacío."**.

> *A la hora de realizar las **comprobaciones** o para mostrar los **mensajes** solo usaremos **literales** en los casos en los que no exista **una constante de clase** destinada para ello. En el apartado **1.1.-Atributos de la clase** se desglosan todas las constantes de clase que se deben definir.*

Si los parámetros son correctos, se procederá a asignar los **valores iniciales de los atributos de objeto** así como a **actualizar los atributos de clase** que sea necesario modificar. En el caso de los **atributos relacionados con la ubicación del dispositivo** (altitud, posición en el eje X y posición en el eje Y), debes tener en cuenta que cuando se instancia un dispositivo se debe crear en su posición inicial que será el origen, por tanto deberás inicializar estos atributos con los **valores a cero**. Para el **atributo relacionado con el tipo de motor**, se deberá asignar su valor por defecto en el caso de que no se corresponda con ninguno de los tres tipos posibles (`Electrico, Combustion o Hibrido`).

2. *Un **constructor sin parámetros**.*

Este constructor generará un objeto `DispositoVolador` con el modelo y el tipo de motor por defecto.

> *Ten en cuenta que **desde los constructores con menos parámetros deberá invocarse a los constructores con más parámetros** utilizando los valores por omisión para completar la inicialización.*

Es fundamental que lo hagas así para **evitar escribir código redundante** que en el futuro podría dar lugar a inconsistencias. Solo en el primer constructor (el que contiene todos los parámetros posibles) deberían hacerse las **comprobaciones** y las **inicializaciones** apropiadas. De este modo se harán siempre de la misma manera y no dependerán del constructor que se esté invocando. El resto de constructores serán simples llamadas a constructores con más parámetros.

*3. Un* ***método "fábrica"*** `crearArrayDispositivos` ***con un parámetro: cantidad de dispositivos****.*

Este método permitirá crear un array de referencias a objetos `DispositivoVolador`, cada una de las cuales apuntará a un nuevo objeto `DispositivoVolador` creado con los parámetros por omisión. En caso de que el parámetro sea menor que 1 o mayor que 20, se lanzaría la excepción `IllegalArgumentException` con el mensaje de error "***Número de dispositivos incorrecto: xxx. Debe ser mayor o igual que 1 y menor o igual que 20***", donde xxx sería la cantidad inválida (un número entero menor que 1 o mayor que 20).

## ¿Cómo pruebo lo que llevo hecho hasta ahora?

Habrás observado que el proceso completo de desarrollo de una clase puede ser bastante largo, siendo necesario escribir cientos (a veces miles) de líneas de código. Para que no tengas que llegar hasta el final y puedas ir probando poco a poco lo que vayas implementando, hemos construido una serie de programas de prueba para cada apartado o grupo de apartados y así podrás ir revisando lo que llevas hecho antes de seguir con la siguiente fase.

Una vez tengas implementada esta primera parte de la clase:

- atributos, tanto de objeto como de clase;
- constructores;

podrías proceder a probar lo que llevas antes de continuar.

Para ello hemos construido un programa llamado `TestEj0102` que se encargará de:

- declarar variables de la clase que has implementado;
- consultar todos los atributos públicos que debería tener la clase, usando los nombres convenidos;
- invocar a los distintos constructores que se deberían haber implementado.

Se realizarán **intentos de llamada con errores** para comprobar si se están lanzando las **excepciones** que se deben lanzar y con los **mensajes de texto** que deben contener, así como **llamadas sin errores** para observar si se ejecutan correctamente y no se producen errores.

El resultado que debería proporcionar el programa por pantalla debería ser el siguiente:

```
------------------------------------------------
CASO DE PRUEBAS 01-02: CONSTRUCTORES Y ATRIBUTOS
------------------------------------------------
------------------------------------------------------------
 - PRUEBA 01 - ATRIBUTOS PÚBLICOS
------------------------------------------------------------
 -> MODELO_DEFECTO: Dispositivo Generico
 -> TIPO_MOTOR_DEFECTO: Electrico
 -> ALTITUD_MAXIMA: 50.0
 -> RANGO_OPERACION: 100

------------------------------------------------------------
 - PRUEBA 02 - CONSTRUCTORES Y FÁBRICA (con datos correctos)
------------------------------------------------------------
Registrando un DispositivoVolador para el dispositivo [D-01,Combustion]
 -> Objeto creado con éxito
Registrando un DispositivoVolador para un dispositivo con los parámetros por def
 -> Objeto creado con éxito

Intentando crear array de 1 objetos.
 -> Array de objetos creado con éxito.
Intentando crear array de 20 objetos.
 -> Array de objetos creado con éxito.

-----------------------------------------------------------
 - PRUEBA 03 - CONSTRUCTORES Y FÁBRICA (con excepciones)
-----------------------------------------------------------
Registrando un DispositivoVolador para el dispositivo [null,Combustion]
 -> Se ha producido un error: El modelo del dispositivo no puede ser nulo.
Registrando un DispositivoVolador para el dispositivo [H-01,]
 -> Se ha producido un error: El tipo de motor no puede estar vacío.

Intentando crear array de -1 objetos.
 -> Se ha producido un error: Número de dipositivos incorrecto: -1. Debe ser may
Intentando crear array de 50 objetos.
 -> Se ha producido un error: Número de dipositivos incorrecto: 50. Debe ser may
```

Si has implementado correctamente y en la forma en que se pide todo lo que se indica en los apartados 1 y 2, **deberías obtener exactamente esta misma salida**. Por tanto, si observas alguna diferencia, antes de continuar implementando métodos, deberías asegurarte de revisar lo que te puede estar fallando. Si continúas desarrollando la clase y tienes errores, luego será mucho más costoso corregirlos.

# 1.3.- Métodos de consulta y actualización

*1. Añade a la clase* ***`DispositivoVolador`*** *los siguientes* ***métodos de consulta*** *(**getters**) para poder obtener la siguiente información de objeto:*

Ronny Overhate (Pixabay License)

1. **`getModelo()`**, que devuelva el **modelo** del dispositivo (**`String`**);
2. **`getTipoMotor()`**, que devuelva el **tipo de motor** del dispositivo (**`String`**);
3. **`getAltitudActual()`**, que devuelva la **altitud actual** del dispositivo (**`double`**);
4. **`getPosicionX()`**, que devuelve la **posición en el eje X** del dispositivo (**`int`**);
5. **`getPosicionY()`**, que devuelve la **posición en el eje Y** del dispositivo (**`int`**).

*2. A estos métodos consultores sobre el estado del objeto habrá que añadir algunos* ***métodos que devuelvan información genérica o global sobre la clase*** *(**estáticos**):*

- **`getNumDispositivos()`**, que devuelva el total de dispositivos creados hasta el momento (**`int`**).

*3. Además de los métodos de consulta, también tenemos que implementar* ***métodos que actualicen la información de los objetos (setters)****:*

- **`setAltitudActual()`**, que modifica la altitud actual del dispositivo. El método recibirá por parámetro la altitud (**`double`**) y no devolverá nada. En el caso de que la altitud no sea correcta (negativa o por encima del máximo) se establecerá la altitud inicial.
- **`setPosicionX()`**, que modifica la posición en el eje X del dispositivo. El método recibirá por parámetro la posición en el eje X (**`int`**) y no devolverá nada. En el caso de que la posición en el eje X no esté dentro del rango de operación permitido se asignará la posición del origen.
- **`setPosicionY()`**, que modifica la posición en el eje Y del dispositivo. El método recibirá por parámetro la posición en el eje Y (**`int`**) y no devolverá nada. En el caso de que la posición en el eje Y no esté dentro del rango de operación permitido se asignará la posición del origen.

## Programa de pruebas

Para probar estos métodos dispones de un programa llamado **`TestEj03`**.

El resultado que debería proporcionar por pantalla este programa de pruebas tendría que ser el siguiente:

```
------------------------------------------------
CASO DE PRUEBAS 03: GETTERS Y MÉTODOS ESTÁTICOS
```

```
------------------------------------------------
------------------------------------------------------------
 - PRUEBA 01 - ATRIBUTOS PÚBLICOS (antes de crear objetos)
------------------------------------------------------------
 -> numDispositivos: 0

------------------------------------------------------------
 - PRUEBA 02 - Creación de objetos y uso de getters()
------------------------------------------------------------
Registrando un DispositivoVolador para el dispositivo [D-01,Electrico]
 -> Objeto creado con éxito
Registrando un DispositivoVolador para el dispositivo [H-01,Combustion]
 -> Objeto creado con éxito
Registrando un DispositivoVolador para el dispositivo [D-02,Hibrido]
 -> Objeto creado con éxito
Array de dispositivos creado con éxito

Leyendo los datos almacenados en el DispositivoVolador
 -> getModelo(): D-01
 -> getTipoMotor(): Electrico
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 0
 -> getPosicionY(): 0

Leyendo los datos almacenados en el DispositivoVolador
 -> getModelo(): H-01
 -> getTipoMotor(): Combustion
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 0
 -> getPosicionY(): 0

Leyendo los datos almacenados en el DispositivoVolador
 -> getModelo(): D-02
 -> getTipoMotor(): Hibrido
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 0
 -> getPosicionY(): 0

------------------------------------------------------------
 - PRUEBA 03 - ATRIBUTOS PÚBLICOS (después de crear objetos)
------------------------------------------------------------
 -> numDispositivos: 3
```

Nuevamente, si has implementado correctamente y en la forma en que se pide todo lo que se indica en este apartado y apartados anteriores, deberías obtener exactamente la misma salida. Todo lo demás debería ser idéntico. Volvemos a recomendarte que si observas alguna diferencia, antes de continuar implementando métodos, revises la implementación de los métodos de este apartado o incluso de métodos anteriores. Si es necesario, repasa incluso la declaración de los atributos. No continúes con la implementación hasta que no observes que todo funciona correctamente. Si no, irás acumulando errores que cada vez serán más difíciles de detectar y corregir.

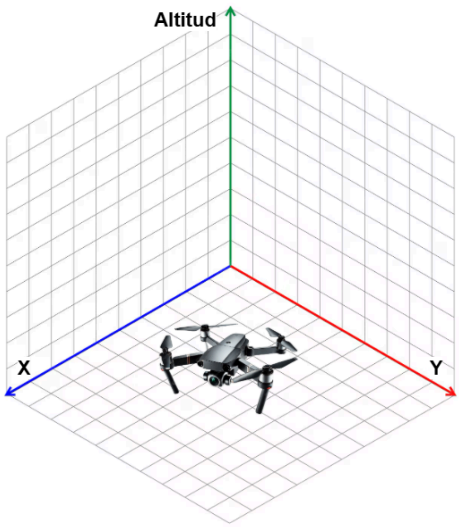
El correcto funcionamiento de los métodos de actualización (***setters***) se comprobará en el test del apartado *1.5. Método toString*.

# 1.4.- Métodos de acción

### *1. Desplazar*

Para poder **desplazar** el dispositivo volador en el plano horizontal habrá que implementar el método `desplazar`. Esto método se encargará de **actualizar** apropiadamente todos **los atributos de estado necesarios** para que se realice adecuadamente el proceso de desplazamiento en los ejes X e Y.

El método `desplazar` recibirá dos **parámetros**: el número de **posiciones** que se desplazará el dispositivo en el **eje X** y el número de **posiciones** que se desplazará en el **eje Y**. Además, este método **no devolverá nada**. Cabe destacar que los números de posiciones que se reciben como parámetro podrán ser tanto valores positivos como negativos ya que si el valor es positivo, el dispositivo avanzará en el eje que corresponda, mientras que si el valor es negativo, el dispositivo retrocederá.




Manuel Marín Peral. *Posición del dispositivo volador*

Las posibles excepciones que podrían lanzarse son:

- si la posición en el eje X al aplicar el desplazamiento recibido como parámetro se sale del rango de operación del dispositivo, la excepción será de tipo `IllegalStateException` y el mensaje de error de la excepción debería ser del tipo **"El dispositivo se ha intentado desplazar fuera de su rango de operación en el eje X."**.
- si la posición en el eje Y al aplicar el desplazamiento recibido como parámetro se sale del rango de operación del dispositivo, la excepción será de tipo `IllegalStateException` y el mensaje de error de la excepción debería ser del tipo **"El dispositivo se ha intentado desplazar fuera de su rango de operación en el eje Y."**.

*Al hacer estas comprobaciones, ten en cuenta que el rango de operación del dispositivo implica el **número máximo de posiciones** que se puede **alejar** en el eje X y en el eje Y **respecto del origen**, tanto hacia delante como hacia atrás.*

### *2. Calcular el consumo*

El método `calcularConsumo` recibirá como parámetro un número entero que indicará la distancia recorrida por el dispositivo, y devolverá un número real con el consumo del dispositivo para dicha distancia recorrida. Si el **parámetro del método** `calcularConsumo` **es incorrecto**, deberá **lanzarse una excepción**. La posible excepción que podría lanzarse es:

- si la distancia recorrida es negativa, la excepción será de tipo `IllegalArgumentException` y el mensaje de error de la excepción debería ser del tipo **"La distancia recorrida no puede ser negativa."**.

wilhei. *Bidones de gasolina* (Pixabay Lincense)

Para realizar el cálculo del consumo deberá comprobarse el tipo de motor del dispositivo, en función de ello tendremos los siguientes casos:

- si el tipo de motor del dispositivo es **"Electrico"**: el consumo del dispositivo será la distancia recorrida multiplicada por el factor **1,2**.
- si el tipo de motor del dispositivo es **"Combustion"**: el consumo del dispositivo será la distancia recorrida multiplicada por el factor **2**.
- si el tipo de motor del dispositivo es **"Hibrido"**: el consumo del dispositivo será la distancia recorrida multiplicada por el factor **1,5**.
- si el tipo de motor del dispositivo es **incorrecto**, se lanzará una excepción de tipo `IllegalStateException` y el mensaje de error de la excepción debería ser del tipo **"Tipo de motor incorrecto."**.

# Programa de pruebas

Para probar estos métodos dispones de un programa llamado `TestEj04`.

El resultado que debería proporcionar por pantalla este programa de pruebas tendría que ser el siguiente:

```
-------------------------------------------------
CASO DE PRUEBAS 04: MÉTODOS DE ACCIÓN
-------------------------------------------------
Registrando un DispositivoVolador para el dispositivo [D-01,Electrico]
 -> Objeto creado con éxito
Registrando un DispositivoVolador para el dispositivo [D-02,Combustion]
 -> Objeto creado con éxito
Registrando un DispositivoVolador para el dispositivo [H-01,Hibrido]
 -> Objeto creado con éxito

------------------------------------------------------------
 - PRUEBA 01 - Uso correcto del método de acción desplazar()
------------------------------------------------------------
Leyendo los datos almacenados en el DispositivoVolador
 -> getModelo(): D-01
 -> getTipoMotor(): Electrico
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 0
 -> getPosicionY(): 0

 -> El dispositivo D-01 se ha desplazado hacia adelante 10 posiciones en el eje X

Leyendo los datos almacenados en el DispositivoVolador
 -> getModelo(): D-01
 -> getTipoMotor(): Electrico
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 10
 -> getPosicionY(): 20

 -> El dispositivo D-01 se ha desplazado hacia detrás 5 posiciones en el eje X y

Leyendo los datos almacenados en el DispositivoVolador
```

```
 -> getModelo(): D-01
 -> getTipoMotor(): Electrico
 -> getAltitudActual(): 0.0
 -> getPosicionX(): 5
 -> getPosicionY(): 15

------------------------------------------------------------------------
 - PRUEBA 02 - Uso incorrecto del método de acción desplazar()
------------------------------------------------------------------------
El dispositivo D-01 se va a intentar desplazar hacia delante 500 posiciones en el

 -> Se ha producido un error: El dispositivo se ha intentado desplazar fuera de s

------------------------------------------------------------------------
 - PRUEBA 03 - Uso correcto del método de acción calcularConsumo()
------------------------------------------------------------------------
Calculando consumos para una distancia recorrida de 100 posiciones
 -> El consumo del dispositivo D-01 es 120,00
 -> El consumo del dispositivo D-02 es 200,00
 -> El consumo del dispositivo H-01 es 150,00

------------------------------------------------------------------------
 - PRUEBA 04 - Uso incorrecto del método de acción calcularConsumo()
------------------------------------------------------------------------
Intentando calcular el consumo del dispositivo D-01 para una distancia recorrida

 -> Se ha producido un error: La distancia recorrida no pue
```

# 1.5.- Método toString

Debes sobrescribir el método `toString` para que devuelva una cadena que represente el **estado del dispositivo** en un momento dado.

Clker-Free-Vector-Images (Pixabay License)

Con el objetivo de mejorar el rendimiento en el tratamiento de cadenas, os pedimos que **utilicéis el método estático `format` de la clase `String` para poder generar toda la cadena de un solo golpe** y de esta manera evitar el tener que ir concatenando poco a poco y generando una enorme cantidad de objetos `String` hasta construir la cadena final que se debe devolver.

El `String` final devuelto por el método debe tener la siguiente estructura:

1. un **corchete de inicio** (carácter `'['`);
2. la etiqueta ***"Modelo="*** junto con el **modelo del dispositivo** y a continuación una coma (",");
3. la etiqueta ***"Tipo de motor="*** junto con el **tipo de motor que tiene el dispositivo** y a continuación una coma (",");
4. la etiqueta ***"Altitud="*** junto con la **altitud a la que se encuentra actualmente el dispositivo** seguida de la etiqueta **"metros"** para indicar la unidad de medida y por último una coma (",");
5. la etiqueta ***"Posicion en X="*** junto con la **posición en el eje X en la que se encuentra actualmente el dispositivo** y a continuación una coma (",");
6. la etiqueta ***"Posicion en Y="*** junto con la **posición en el eje Y en la que se encuentra actualmente el dispositivo** y a continuación una coma (",");
7. un **corchete de fin** (el carácter `']'`).

Analicemos un posible ejemplo de salida:

```
[Modelo=D-01, Tipo de motor=Electrico, Altitud=5,00 metros, Posicion en X=10, Posicion
                                       en Y=20]
```

donde observamos que:

- el **modelo** del dispositivo es **D-01**;
- el **tipo de motor** del dispositivo es **"Electrico"**;
- la **altitud actual** del dispositivo es de **5,00** metros;
- la **posición en el eje X** del dispositivo es **10**;
- la **posición en el eje Y** del dispositivo es **20**.

Aquí tienes algunos ejemplos adicionales de posibles valores devueltos por `toString`:

```
[Modelo=D-01, Tipo de motor=Electrico, Altitud=0,00 metros, Posicion en X=0, Posicion en Y=0]
[Modelo=D-01, Tipo de motor=Electrico, Altitud=25,00 metros, Posicion en X=0, Posicion en Y=30]
```

Como puedes observar, los ejemplos anteriores muestran el mismo objeto en dos estados diferentes:

1. A una altitud de 0,00 metros y en la posición 0 tanto en el eje X como en el Y.
2. A una altitud de 25,00 metros, en la posición 0 del eje X y en la posición 30 del eje Y.

# Programa de pruebas

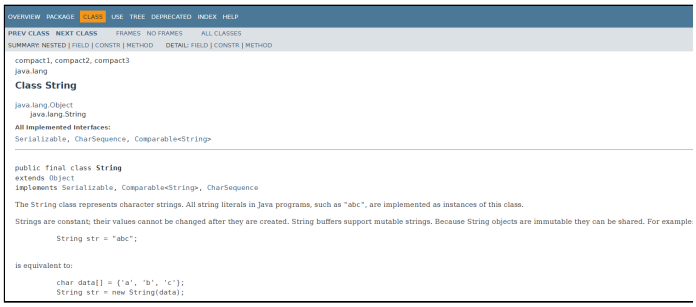
Para probar este método dispones de un programa llamado **`TestEj05`**.

El resultado que tendría que proporcionar por pantalla este programa de pruebas debería ser el siguiente:

```
----------------------------------------
CASO DE PRUEBAS 05: método toString()
----------------------------------------------

Registrando un DispositivoVolador para el dispositivo [D-01,Electrico]
 -> Objeto creado con éxito

------------------------------------------------------------
 - PRUEBA 01 - Visualización del estado del objeto
------------------------------------------------------------
 -> El dispositivo D-01 tiene los siguientes datos iniciales
Datos del dispositivo D-01:  [Modelo=D-01, Tipo de motor=Electrico, Altitud=0,00

 -> El dispositivo D-01 se ha desplazado hacia adelante 50 posiciones en el eje
Datos del dispositivo D-01:  [Modelo=D-01, Tipo de motor=Electrico, Altitud=0,00

 -> El dispositivo D-01 modifica su altitud actual a 25 metros y su posicion en
Datos del dispositivo D-01:  [Modelo=D-01, Tipo de motor=Electrico, Altitud=25,0

 -> El dispositivo D-01 intenta establecer su altitud actual a 250 metros y su p
Datos del dispositivo D-01:  [Modelo=D-01, Tipo de motor=Electrico, Altitud=0,00
```

# 1.6.- Documentación javadoc

Debes documentar adecuadamente el código con **comentarios javadoc** para poder generar una documentación útil y que siga las convenciones. Te recordamos la **guía-resumen** que ya incluimos en los contenidos:




*Documentación javadoc* (Dominio público)

- **documentación general de la clase**. Debes incluir una descripción lo más completa y detallada posible de la clase, explicando todo aquello que consideres que pueda ser de interés para otros programadores que vayan a utilizarla. Ten en cuenta que no tienen por qué conocer cómo está implementada por dentro, sino únicamente cómo utilizarla (cómo usar sus miembros que actúan como interfaz con el código externo: métodos y atributos públicos). Esta descripción se puede subdividir en:
  - **descripción corta** o resumen de la clase. Consiste en todo el texto que incluyas hasta el primer punto (".") que contenga el texto. Es el texto que aparecerá en la tabla resumen de clases del paquete en el que se encuentre la clase;
  - **descripción larga** de la clase. Todo el texto que haya después del primer punto. Ambas descripciones aparecerán en la descripción general de la clase;
- **documentación de atributos públicos y protegidos** (los miembros protegidos los verás en la próxima unidad) indicando para cada uno de ellos una pequeña descripción;
  - si se trata de un **atributo constante**, incluye su valor usando la etiqueta `@value`. **Debes usar la etiqueta `@value` y no escribir un literal en el comentario javadoc**. De esta manera si se modificara el valor de la constante en el código, no habría que modificar el comentario.
- **documentación de los métodos públicos y protegidos**, incluyendo para cada método:
  - **descripción del método**:
    - **descripción corta** o resumen del método. Consiste en todo el texto que incluyas hasta el primer punto. Es el texto que aparecerá en la tabla resumen de métodos;
    - **descripción larga** del método. Todo el texto que haya después del primer punto. Ambas descripciones aparecerán en la descripción detallada del método;
  - **descripción del valor devuelto** (etiqueta `@return`), si es que devuelve algo. Esta información aparecerá en la descripción detallada del método;
  - **descripción de cada uno de los parámetros** (etiqueta `@param`);
  - **descripción de las posibles excepciones** que puede lanzar (etiqueta `@throws`);
- **documentación de los constructores públicos**, de manera similar a los métodos, salvo que nunca tendrán etiqueta `@return`.

**En el apartado información de interés te proporcionamos una muestra de cómo podría quedar tu documentación *javadoc*.**

Tus descripciones y comentarios no tienen por qué coincidir exactamente con los que se incluyen en ese modelo, pero puede servirte de guía si no tienes claro cómo enfocar tu documentación. Basándote en ese modelo y en los textos de la descripción de la tarea deberías tener más que suficiente material para elaborar una documentación detallada. En esta tarea el objetivo es que aprendas a colocar correctamente los comentarios javadoc más que el propio texto que escribas, que lo puedes copiar del modelo que se te proporciona.

**Es importante que una vez que hayas incluido todos tus comentarios *javadoc* en el código pruebes a generar la documentación javadoc (acción "*Generar Javadoc*"**

**en NetBeans) y le eches un vistazo. Es habitual haber cometido pequeños fallos en la redacción de la documentación y los olvidos suelen ser frecuentes (por ejemplo parámetros o excepciones sin documentar).**

# 2.- Información de interés

## Recursos necesarios y recomendaciones

mohamed Hassan (Licencia Pixabay)

- Se proporciona un proyecto base sobre el que debes trabajar: ProyectoBase. En él tendrás incluidos los programas de prueba (cada uno con su método `main`), así como un conjunto de utilidades para la prueba y el esqueleto de tu clase `DispositivoVolador`. **Es obligatorio utilizar este proyecto y sus programas de pruebas** . La mayoría de estos programas de prueba aparecerán con errores de compilación, pues están intentando utilizar métodos y atributos de la clase `DispositivoVolador` que aún no has implementado. Conforme vayas desarrollando tu clase `DispositivoVolador`, esos errores deberán ir desapareciendo.
- Es fundamental **evitar la redundancia de código** (y de ese modo minimizar las posibles inconsistencias), siempre que sea posible, en aquellos fragmentos de código que realicen acciones similares, especialmente en los **constructores y métodos sobrecargados**. En esos casos debes aprovechar la llamada desde un método a otro si tienes que ejecutar una misma acción (comprobaciones, inicializaciones, actualizaciones, etc.) para evitar tener que escribir el mismo código más de una vez (redundancia). Una de las ventajas de la programación orientada a objetos es precisamente esa.
- Las **excepciones** que hay que lanzar desde algunos métodos o constructores de la clase `DispositivoVolador` y que se capturan desde el programa principal son `IllegalArgumentException`, `IllegalStateException` y `NullPointerException`. Dependiendo del método y de la circunstancia que se produzca, algunos puede que lancen una de ellas o las tres. ¡No lo olvides!
- Se pide utilizar el método estático `String.format` para implementar internamente el método `toString` . La idea es que, en lugar de concatenar una cadena tras otras para obtener el resultado final, se realice la creación y formateo completo de la cadena de salida de un solo golpe mediante el uso de este método.
- Aunque en cada apartado se te ha proporcionado cuál debería ser la salida del programa de prueba, te ofrecemos aquí la posibilidad de descargar en un único archivo PDF la salida de todos esos programas, por si quieres tener toda esa información recogida en un único documento. Como ya se ha dicho en más de una ocasión, **deberías comprobar que el uso de los programas de prueba con tu clase `DispositivoVolador` genera los mismos resultados** (salvo los relacionados con la fecha de realización de las pruebas). Si no es así, deberías plantearte que es probable que hayas cometido algunos errores o incorrecciones:
  - Archivo PDF con la salida del programa al probar todos esos casos: Resultado esperado en los programas de prueba (pdf - 20 KB)
- Te proporcionamos una muestra de cómo podría quedar tu **documentación javadoc**. Tus descripciones y comentarios no tienen por qué coincidir exactamente con estos, pues no es más que un ejemplo, pero puede servirte de guía si no tienes claro cómo enfocar tu documentación (está directamente basada en los textos de la descripción de la tarea): Modelo de javadoc (zip - 112 KB) .

# Indicaciones de entrega

Una vez completados todos los ejercicios de la tarea, el envío se realizará a través de la plataforma. Comprime la carpeta del proyecto NetBeans en un fichero .zip y nómbralo siguiendo las siguientes pautas:

***Apellido1_Apellido2_Nombre_PROG05_Tarea***

Recuerda que **tu proyecto NetBeans también debe llamarse así** (tanto la carpeta donde se encuentra el proyecto como el nombre del proyecto en Netbeans tienes que renombrarlos respecto al nombre original del proyecto base que vas a utilizar). **Si no lo haces, todos los proyectos se llamarán igual y la corrección de tu tarea podría confundirse con la de otra persona**.

Por tanto, **si no entregas tu tarea siguiendo estas reglas, se te devolverá para que lo hagas correctamente** .

# 2.1.- Programas de pruebas

Bluesnap (Pixabay Lincense)

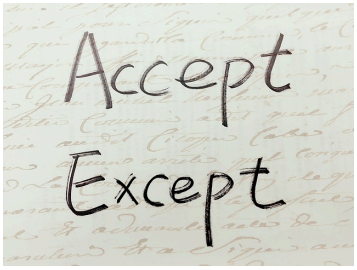
Como ya se ha comentado anteriormente, el proyecto base de esta tarea contiene una serie de **programas de pruebas** (clases `TestEj0102`, `TestEj03`, etc.) para probar el funcionamiento de la clase `DispositivoVolador` paso a paso. Se hacen pruebas de creación de dispositivos válidos así como de dispositivos con valores erróneos y también se intentan realizar operaciones (llamadas a métodos) que dan lugar a errores así como operaciones que deberían poder efectuarse correctamente. Toda la información que se recoja de las acciones que se intenten llevar a cabo, tanto si tienen éxito como si no, se irá mostrando en consola y podrás ir observando si el comportamiento de tu clase es el apropiado respecto a las especificaciones proporcionadas por el enunciado de la tarea.

Podrás **comprobar si tu clase tiene el comportamiento apropiado si la salida obtenida por los programas de prueba al usar tu clase coincide con los ejemplos de salida que se te proporcionan en cada paso de la tarea**. Si observas alguna diferencia deberás revisar la implementación de tu clase para intentar localizar en qué has podido confundirte o qué se te ha olvidado hacer.

**Los programas de pruebas están completos. No debería modificarse ni una sola línea**. Los ha desarrollado otro programador utilizando las mismas especificaciones que habéis recibido vosotros para implementar la clase. Únicamente falta probarlo con vuestra implementación de la clase `DispositivoVolador` para certificar que se ha codificado lo que se ha pedido de la forma que se ha indicado en estas instrucciones.

Es **imprescindible** que emplees estas herramientas de prueba que se te proporcionan. De esta manera todos tendréis que amoldaros obligatoriamente a las interfaces definidas para la clase `DispositivoVolador`. Es decir, que vuestra clase `DispositivoVolador` deberá encajar en estos programas como si fuera una pieza más de un **puzzle**. Cada uno habrá implementado internamente los atributos y los métodos de la manera que haya considerado más apropiada (nombres de atributos privados, variables locales, cálculos internos, uso de unas u otras estructuras de control, etc.), pero todos habréis tenido que definir la **misma interfaz pública de entrada/salida** (nombre de la clase, nombres de los atributos públicos, nombres de los métodos públicos, parámetros de los métodos, valores devueltos, excepciones lanzadas, etc.) para que pueda ser compatible con estos programas de prueba. La interfaz de cada método la podréis encontrar detalladamente descrita en la **documentación javadoc** que se os ha proporcionado como modelo.

# 2.2.- Excepciones




JiaqiLi Exception (Pixabay License)

Uno de los puntos clave en esta tarea se encuentra en la **comprobación y gestión de posibles errores** en los métodos de la clase que tienes que implementar. Eso significa que debes prestar atención a cada uno de los posibles fallos que se detallan en la descripción de los métodos que hemos hecho en los apartados anteriores.

Según el tipo de error que se pueda producir, el método tendrá que lanzar (`throw`) una excepción del tipo que se especifique en la descripción del método incluyendo el **texto de error** que se indique en cada caso. Además, el método deberá incluir en su cabecera la cláusula `throws` con la excepción o excepciones que puede lanzar.

En algunos casos, la herramienta NetBeans puede ayudarte a generar automáticamente métodos que aún no has implementado. Debes tener cuidado con esto, pues es posible que no se genere exactamente lo que realmente necesitas y que además falten los `throws` de la cabecera. El esqueleto de los métodos debe ser escrito por vosotros, no de manera automática por la herramienta. No es admisible que en el código de la clase que presentéis como solución a la tarea incluya cosas como:

```
    String getModelo() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
    }
```

**Eso, además de quedar bastante mal en el código de tu clase, será penalizado en la corrección.**

# 3.- Evaluación de la tarea

## Criterios de evaluación implicados

Del RA4 (*Desarrolla programas organizados en clases analizando y aplicando los principios de la programación orientada a objetos*):

- a) Se ha reconocido la sintaxis, estructura y componentes típicos de una clase.
- b) Se han definido clases.
- c) Se han definido propiedades y métodos.
- d) Se han definido métodos estáticos.
- e) Se han creado constructores.
- f) Se han utilizado mecanismos para controlar la visibilidad de las clases y de sus miembros.
- g) Se han desarrollado programas que instancien y utilicen objetos de las clases creadas anteriormente.
- h) Se han definido y utilizado clases heredadas.
- i) Se han creado y utilizado conjuntos y librerías de clases.


## ¿Cómo valoramos y puntuamos tu tarea?

Cada uno de los CE cubiertos por esta tarea será evaluado a través de los criterios de corrección y puntuación indicados detalladamente en la rúbrica de la tarea que puedes consultar a continuación:

| ÍTEMS | PUNTUACIONES | | | | | |
|---|---|---|---|---|---|---|
| (RA4.a, RA4.b, RA4.c, RA4.d, RA4.f, RA4.g) APARTADO 1: Declaración de los atributos de clase. | No se realiza correctamente. **0 puntos** | Casi no se declaran atributos de clase para la clase DispositivoVolador de manera correcta **0.1 puntos** | Se declaran muy pocos atributos de clase para la clase DispositivoVolador de manera correcta. **0.2 puntos** | Se declaran algunos atributos de clase para la clase DispositivoVolador de manera correcta. **0.3 puntos** | Se declaran la mayoría de los atributos de clase para la clase DispositivoVolador de manera correcta **0.4 puntos** | Se declaran todos los atributos de clase para la clase DispositivoVolador de manera correcta. **0.5 puntos** |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 1: Declaración de los atributos de objeto inmutables. | No se realiza correctamente. **0 puntos** | | Se declaran algunos atributos de objeto inmutables para la clase DispositivoVolador de manera correcta. **0.15 puntos** | | Se declaran todos los atributos de objeto inmutables para la clase DispositivoVolador de manera correcta. **0.3 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 1: Declaración de los atributos de objeto variables. | No se realiza correctamente. **0 puntos** | Casi no se declaran atributos de objeto variables para la clase DispositivoVolador de manera correcta. **0.1 puntos** | | Se declaran algunos atributos de objeto variables para la clase DispositivoVolador de manera correcta. **0.2 puntos** | | Se declaran todos los atributos de objeto variables para la clase DispositivoVolador de manera correcta. **0.3 puntos** |
| (RA4.a, RA4.b, RA4.c, RA4.e, RA4.f, RA4.g) APARTADO 2: Constructor con parámetros. Comprobación de que los parámetros no sean nulos. Si no es así, se lanza una excepción *NullPointerException* con un mensaje de error apropiado. | No se realiza correctamente. **0 puntos** | | Comprueba que los parámetros no sean nulos con algún error. **0.1 puntos** | | Comprueba que los parámetros no sean nulos de manera correcta. **0.2 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.e, RA4.f, RA4.g) APARTADO 2: Constructor con parámetros. Comprobación de que los parámetros no sean incorrectos. Si no es así, se lanza una excepción *IllegalArgumentException* con un mensaje de error apropiado. | No se realiza correctamente. **0 puntos** | | Comprueba que los parámetros no sean incorrectos con algún error. **0.1 puntos** | | Comprueba que los parámetros no sean incorrectos de manera correcta. **0.2 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.e, RA4.f, RA4.g) APARTADO 2: Constructor con parámetros. Inicialización de los atributos de objeto, tanto inmutables como de estado. | No se realiza correctamente. **0 puntos** | | Inicializa los atributos de objeto con algún error. **0.2 puntos** | | Inicializa los atributos de objeto de manera correcta. **0.4 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.e, RA4.f, RA4.g, RA4.h) APARTADO 2: Constructor por defecto. Asignación de los valores por omisión a los atributos de objeto. | No se realiza correctamente. **0 puntos** | | Asigna por omisión los atributos de objeto con algún error. **0.1 puntos** | | Asigna por omisión los atributos de objeto de manera correcta. **0.2 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.d, RA4.e, RA4.f, RA4.g) APARTADO 2: Método fábrica. Comprobación de que el argumento es válido. Si no es así, se lanza una excepción *IllegalArgumentException* con un mensaje de error apropiado. | No se realiza correctamente. **0 puntos** | | | Se realiza la comprobación de manera correcta. **0.1 puntos** | | |

| | | | | | |
|---|---|---|---|---|---|
| (RA4.a, RA4.b, RA4.c, RA4.d, RA4.e, RA4.f, RA4.g) APARTADO 2: Método fábrica. Creación e inicialización del array de objetos DispositivoVolador. | No se realiza correctamente.<br>**0 puntos** | Se realiza la creación e inicialización del array de objetos DispositivoVolador con algún error.<br>**0.2 puntos** | Se realiza la creación e inicialización del array de objetos DispositivoVolador de manera correcta.<br>**0.4 puntos** | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 3: Métodos de consulta. Implementación correcta de los métodos getter. | No se realiza correctamente.<br>**0 puntos** | Se implementan los métodos getter con muchos errores.<br>**0.2 puntos** | Se implementan los métodos getter con algunos errores.<br>**0.3 puntos** | Se implementan los métodos getter con pocos errores.<br>**0.4 puntos** | Se implementan los métodos getter de manera correcta.<br>**0.5 puntos** |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 3: Métodos de consulta. Implementación correcta de los métodos getter estáticos para la devolución de información de la clase. | No se realiza correctamente.<br>**0 puntos** | Se implementan los métodos getter estáticos con algún error.<br>**0.15 puntos** | Se implementan los métodos getter estáticos de manera correcta.<br>**0.3 puntos** | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 3: Métodos de actualización. Implementación correcta de los métodos setter. | No se realiza correctamente.<br>**0 puntos** | Se implementan los métodos setter con muchos errores.<br>**0.25 puntos** | Se implementan los métodos setter con algún error.<br>**0.5 puntos** | Se implementan los métodos setter de manera correcta.<br>**0.75 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 4: Desplazar. Se define la cabecera del método desplazar correctamente: nombre, modificadores, visibilidad, valor devuelto, posibles parámetros, excepciones (throws)… | No se realiza correctamente.<br>**0 puntos** | Se define parte de la cabecera de los métodos correctamente.<br>**0.1 puntos** | Se define la cabecera de los métodos correctamente.<br>**0.2 puntos** | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 4: Desplazar. Se lanzan las excepciones necesarias, con un mensaje de error apropiado, para el control de parámetros de entrada en el método desplazar. | No se realiza correctamente.<br>**0 puntos** | Se lanzan pocas excepciones correctamente.<br>**0.25 puntos** | Se lanzan algunas de las excepciones necesarias correctamente.<br>**0.5 puntos** | Se lanzan la mayoría de excepciones necesarias correctamente.<br>**0.75 puntos** | Se lanzan todas las excepciones necesarias correctamente.<br>**1.2 puntos** |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g) APARTADO 4: Se actualizan los atributos de objeto necesarios en el método desplazar. | No se realiza correctamente.<br>**0 puntos** | Se actualizan los atributos de objeto con muchos errores.<br>**0.25 puntos** | Se actualizan los atributos de objeto con algún error.<br>**0.5 puntos** | Se actualizan los atributos de objeto correctamente.<br>**0.75 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g, RA4.i) APARTADO 4: Calcular consumo. Se lanzan las excepciones necesarias, con un mensaje de error apropiado, para el control de parámetros de entrada en el método de calcular consumo. | No se realiza correctamente.<br>**0 puntos** | Se lanzan algunas de las excepciones necesarias correctamente.<br>**0.3 puntos** | Se lanzan todas las excepciones necesarias correctamente.<br>**0.7 puntos** | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g, RA4.i) APARTADO 4: Calcular consumo. Se realizan los cálculos necesarios para calcular el consumo. | No se realiza correctamente.<br>**0 puntos** | Se realizan los cálculos necesarios con muchos errores.<br>**0.35 puntos** | Se realizan los cálculos necesarios con algún error.<br>**0.75 puntos** | Se realizan los cálculos necesarios correctamente.<br>**1 puntos** | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g, RA4.i) APARTADO 5: Método toString(). Se incluyen en la cadena de salida todas las características y valores de estado solicitados. | No se realiza correctamente.<br>**0 puntos** | Se incluyen en la cadena de salida algunas características y valores de estado.<br>**0.25 puntos** | Se incluyen en la cadena de salida todas las características y valores de estado.<br>**0.5 puntos** | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g, RA4.i) APARTADO 5: Método toString(). Se respeta el formato solicitado, devolviendo la información de forma correcta. | No se realiza correctamente.<br>**0 puntos** | Se respeta el formato solicitado.<br>**0.25 puntos** | | | |
| (RA4.a, RA4.b, RA4.c, RA4.f, RA4.g, RA4.i) APARTADO 5: Método toString(). Se ha utilizado el método String.format() para la construcción de la cadena de salida. | No se realiza correctamente.<br>**0 puntos** | Se ha utilizado el método String.format().<br>**0.25 puntos** | | | |
| (RA4.a, RA4.b, RA4.f, RA4.h, RA4.i) APARTADO 6: Documentación javadoc. Se documentan apropiadamente los atributos constantes. Uso de @value. | No se realiza correctamente.<br>**0 puntos** | Se han documentado los atributos correctamente.<br>**0.2 puntos** | | | |
| (RA4.a, RA4.b, RA4.f, RA4.h, RA4.i) APARTADO 6: Documentación javadoc. Se documentan apropiadamente los constructores. | No se realiza correctamente.<br>**0 puntos** | Se han documentado los constructores correctamente.<br>**0.2 puntos** | | | |

| | | |
|---|---|---|
| (RA4.a, RA4.b, RA4.f, RA4.h, RA4.i) APARTADO 6: Documentación javadoc. Se documentan apropiadamente los getters y setters. | No se realiza correctamente. **0 puntos** | Se han documentado los getters y setters correctamente. **0.2 puntos** |
| (RA4.a, RA4.b, RA4.f, RA4.h, RA4.i) APARTADO 6: Documentación javadoc. Se documentan apropiadamente los métodos de acción. | No se realiza correctamente. **0 puntos** | Se han documentado los métodos de acción correctamente. **0.2 puntos** |
| (RA4.a, RA4.b, RA4.f, RA4.h, RA4.i) APARTADO 6: Documentación javadoc. Se documenta apropiadamente el método toString(). | No se realiza correctamente. **0 puntos** | Se ha documentado el método toString() correctamente. **0.2 puntos** |

Tanto en la tarea en general como en la actividad que tenga que desarrollar en particular, se tendrán en cuenta los siguientes puntos de control o elementos evaluables a la hora de corregir tu tarea:

- Se ha creado un único proyecto con todos los ejercicios en él utilizando **el entorno NetBeans 16 y Java Development Kit 17 (JDK)**.
- **Las clases y los programas de prueba deben compilar.** Cualquier funcionalidad pedida en el enunciado debe poderse probar y ha de funcionar correctamente de acuerdo a las especificaciones del enunciado.
- **La ejecución de los programas de prueba que usan la clase no debe romperse**. Si eso sucede es porque la clase no ha sido implementada correctamente con respecto a los requisitos indicados en el enunciado.
- Se sigue la **estructura** propuesta de **declaración de variables**, **entrada de datos**, **procesamiento** y **salida de resultados**, tal y como se indica en la plantilla y el enunciado.
- Se declaran las **variables necesarias** con el **tipo adecuado** y con **nombres apropiados**.
- Se declaran **identificadores** con **nombres significativos o descriptivos** que representen de alguna manera la información que están almacenando para que **el código quede lo más claro, legible y autodocumentado posible**. Los **identificadores** cumplen con el **convenio** sobre **"*asignación de nombres a identificadores*"** establecido para el lenguaje Java para constantes, variables, métodos, clases, etc.
- No se incluyen **elementos superfluos** ni **innecesarios**, ni se llevan a cabo **operaciones sin sentido**.
- No se utilizan **estructuras de salto incondicional para el control del flujo**, o bien herramientas como return, System.exit() o cualquier otro mecanismo que no sea el fin de la ejecución natural del programa.
- Se llevan a cabo los **cálculos y comprobaciones necesarias** para llevar a cabo las tareas que se pide en cada actividad. Para ello habrá que construir **expresiones apropiadas** y hacer uso de los **operadores correctos**.
- El código de los programas está correctamente **indentado**. Es fundamental para poder observar e intuir rápidamente, y de forma visual, la estructura de los programas. Recuerda que NetBeans puede hacer esto por ti (selecciona el código a "formatear" o "embellecer" y pulsa Alt+Mayús.+F).
- El código está apropiadamente **comentado**. Se observa una **corrección ortográfica y gramatical**, así como la **coherencia en las expresiones lingüísticas** en los comentarios incluidos.
- En los programas **se muestra la información esperada y correcta por pantalla en un formato apropiado**. Se observa una **corrección ortográfica y gramatical,** así como la coherencia en las expresiones lingüísticas, tanto en los textos de los mensajes que aparezcan en pantalla para pedir información de entrada al usuario como a la hora de mostrar resultados de salida. Se evitan mensajes de entrada de datos y/o salida de resultados inapropiados, descontextualizados, insuficientes o incorrectos.

Aparte de todos estas cuestiones de carácter "técnico", también se valorará en cada uno de los apartados:

- que la respuesta sea personal y no copiada de Internet o de otras fuentes;
- que sea coherente con el resto de respuestas y con los conceptos de la unidad;
- que muestre que se comprenden adecuadamente los conceptos tratados.

## IMPORTANTE: Motivos de devolución de la tarea o de calificación mínima

La tarea obtendrá automáticamente una calificación de **cero** si:

- No se entrega **un único proyecto** que incluya los ejercicios que se piden en la tarea utilizando el IDE NetBeans.
- Alguno de los programas presenta **errores de compilación** impidiendo su ejecución y prueba.
- **No se respeta el proyecto base** que se entrega al alumno/a para realizar la tarea.
- **Se modifican los programas de prueba** proporcionados en el proyecto base.
- Se utilizan **contenidos** en alguno de los ejercicios del proyecto **que no se han estudiado** en la unidad actual o anteriores a la misma.
- Se presenta un proyecto total o parcialmente copiado de otro alumno/a.

**Recordatorio**: Si una tarea obtiene calificación cero por alguno de los motivos previamente mencionados, será devuelta al alumno/a. Este tendrá derecho a una segunda entrega únicamente en los siguientes casos:

- Que se trate de la primera vez que presenta la tarea.
- Que la fecha de entrega inicial haya sido al menos una semana antes de la fecha límite establecida para dicha tarea.