Tarea online

Título de la tarea: Utilización avanzada de clases.

Unidad: 06

Ciclo formativo y módulo: DAM/DAW, Programación.

Curso académico: 2025/26

¿Qué contenidos o resultados de aprendizaje trabajaremos?

Esencialmente, trabajaremos con el siguiente resultado de aprendizaje:

- **RA7**.-*Desarrolla programas aplicando características avanzadas de los lenguajes orientados a objetos y del entorno de programación*.

1.- Descripción de la tarea

Caso práctico

La empresa que gestionaba los dispositivos voladores de la tarea anterior ha ampliado su operativa y ahora gestiona más tipos de dispositivos, por lo que ha encargado a **BK Programación** la implantación de algún mecanismo que permita llevar un control no solo de los dispositivos voladores genéricos como hasta ahora, sino que añade la gestión de drones orientados al reparto y a carreras.

FrederickSnr (Dominio público)

Para ello es preciso crear una clase padre de dispositivos voladores para prepararse ante futuras ampliaciones. Además, se deben crear dos subclases de esta, que serán dron de reparto y dron de carreras, que permitan gestionar estos tipos de dispositivos voladores y guardar información de los mismos.

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 la clase padre y la subclase del dron de reparto.

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

- Los atributos comunes de los dispositivos voladores deben pertenecer a la superclase llamada "**DispositivoVolador**" y los atributos específicos del dron de reparto serán propios, como la **capacidad máxima** y la **capacidad actual**.

- Como hay varios métodos que se encargan del reparto, se debería crear una interfaz "**Repartible**" para establecer estos métodos.

- Además, como los dispositivos voladores tienen la capacidad de desplazarse se creará una interfaz "**Desplazable**" para establecer el método que defina el comportamiento de desplazar.

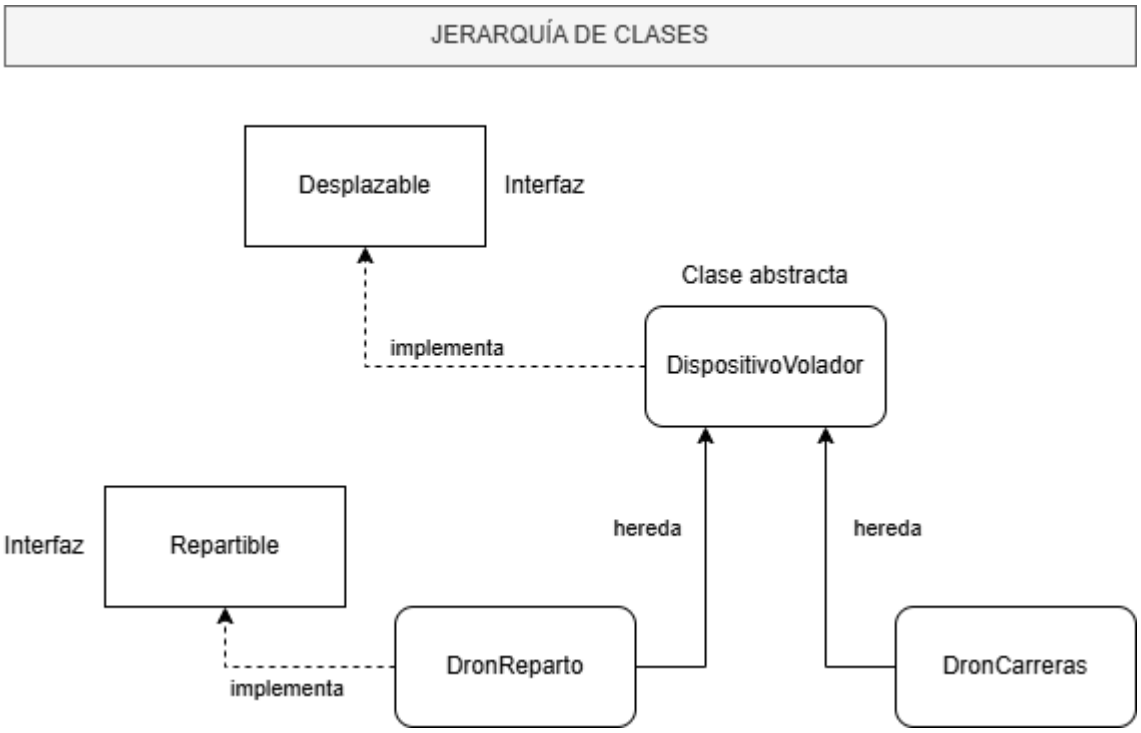
¿Qué te pedimos que hagas?

La tarea consiste en la implementación de una **jerarquía de clases basada en los tipos de dispositivos voladores**. Nos basaremos en la clase abstracta `DispositivoVolador` y, a partir de

ella, se implementarán dos serie de clases derivadas (`DronReparto y DronCarreras`) con unas características (atributos) y comportamientos (métodos) específicos. De esta manera, practicaremos el uso de una gran parte de los mecanismos y recursos relacionados con el **desarrollo avanzado de clases** estudiados en esta unidad (herencia y composición de clases, redefinición y ampliación de métodos, implementación de interfaces, polimorfismo, ligadura dinámica, etc.).

Algunas de las clases que tendrás que implementar serán **abstractas** (no se podrán instanciar objetos a partir de ellas, tan solo implementar otras subclases derivadas a partir de ellas) mientras que otras sí serán **instanciables**. Por otro lado, se te pedirá que definas dos **interfaces** para especificar, en una de ellas, los métodos que se encarguen del desplazamiento, y en la otra, los encargados del reparto.

La jerarquía de clases debe quedar tal que así:



A lo largo de los siguientes apartados de la tarea se irán explicando detalladamente cada uno de los elementos (clases e interfaces) que tendrás que implementar, proporcionándote toda la información que necesitas para que puedas desarrollarlos sin dificultad.

1.1.1.- Atributos

En la clase `DispositivoVolador` se definen los atributos mínimos para representar las características elementales de cualquier **dispositivo volador.**

> ***Todos los atributos*** *deben ser* ***accesibles únicamente por la clase*** `DispositivoVolador` ***y por sus clases hijas****, exceptuando las* ***constantes****, que deben ser públicas tal y como se indica más abajo.*

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. Sólo se podrá acceder a estos atributos desde la clase** `DispositivoVolador`**.**

- **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. **Se podrá acceder a estos atributos desde cualquier clase que herede de la clase** `DispositivoVolador`**.**

- 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. No olvides que también puedes ver la información en el* ***javadoc*** *que adjuntamos a la tarea.*

Además, también se dispondrá de otros atributos que pertenecerán a la clase (atributos de clase), más que a un objeto en particular, compartiendo su valor para todos los objetos. Estos atributos tendrán sentido siempre, independientemente de que existan o no instancias de la clase (**atributos estáticos**) y se accederá a ellos a través del nombre de la clase (y no de una referencia a un objeto). **Sólo se podrá acceder a estos atributos desde la clase** `DispositivoVolador`**.**

- el **número de dispositivos** creados hasta ese momento.

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 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 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.*

1.1.2.- Constructores

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

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 1.1.1.-Atributos 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.

1.1.3.- Métodos

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

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. Además de esos métodos consultores sobre el estado del objeto, también habrá que implementar algún ***método que devuelva información genérica o global sobre la clase (estático):***

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

3. También tendremos que implementar ***métodos para modificar atributos de nuestra clase (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.

4. Además implementaremos el ***método de acción*** *relacionado con la interfaz* **`Desplazable`**:

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.

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.*

*5. Además implementaremos un **método de acción para calcular el consumo** de los dispositivos voladores:*

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 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."**.

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."**.

Este método será sobreescrito en la subclase `DronCarreras`.

*6. También implementaremos un **método abstracto para realizar el mantenimiento** de los dispositivos voladores:*

Para realizar el mantenimiento de cada dispositivo volador implementaremos el siguiente método:

- `realizarMantenimiento()`, que servirá para realizar el mantenimiento del dispositivo volador. Este método no tendrá ningún parámetro y no devolverá nada.

Este método será sobreescrito en cada subclase.

7. Por último, sobrescribiremos el método `toString` para que devuelva una cadena que represente el estado del dispositivo volador en un momento dado

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:

- la etiqueta ***"Modelo="*** junto con el **modelo del dispositivo** y a continuación una coma (",");
- la etiqueta ***"Tipo de motor="*** junto con el **tipo de motor que tiene el dispositivo** y a continuación una coma (",");
- 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 (",");
- 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 (",");
- 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 (",").

1.2.- Interfaces Desplazable y Repartible

Algunas subclases de la clase **`DispositivoVolador`** tendrán que implementar las interfaces que se describen a continuación:

1.- Interfaz `Desplazable`

La interfaz **`Desplazable`** será implementada por cualquier dispositivo volador. Para proporcionar esta funcionalidad habrá que escribir el siguiente método:

- método **`desplazar()`**, que permite realizar el desplazamiento del dispositivo volador en los ejes X e Y.

Las características de este método de la interfaz son las mismas que se indicaron para la tarea 5.

Todos los detalles de este método los puedes ver en el apartado 1.1.3.

2.- Interfaz `Repartible`

La interfaz **`Repartible`** será implementada por cualquier dispositivo volador capaz de cargar y repartir paquetes (dron de reparto). Para proporcionar esta funcionalidad habrá que escribir los siguientes métodos:

- método **`cargar`**(), que permite cargar paquetes en un dron de reparto.
- método **`repartir`**(), que permite repartir paquetes de un dron de reparto.

Todos los detalles de estos métodos los puedes ver en el apartado 1.3.3.

1.3.- Clase DronReparto

Esta clase hereda de la **superclase** `DispositivoVolador`.

En esta clase definiremos los atributos mínimos para representar las características adicionales que tiene un dron de reparto con respecto a un dispositivo volador genérico.

Esta clase **sí que debe ser instanciable**, ya que sí crearemos objetos de este tipo.

Esta clase **no debe ser heredable**, es decir, ninguna otra clase puede heredar de `DronReparto`.

Además, esta clase implementa la interfaz `Repartible`.

User6702303 (Dominio público)

La clase `DronReparto` debe tener sus **atributos**, **constructores** y **métodos** que describiremos en los siguientes puntos.

1.3.1.- Atributos

Esta clase debe heredar todos sus atributos de la clase **`DispositivoVolador`**, teniendo en cuenta lo siguiente:

> ***Todos los atributos*** *deben ser* ***accesibles únicamente por la clase*** **`DronReparto`**.

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**:

- **capacidad máxima** que puede soportar el dron (en paquetes).

> *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, estará el siguiente atributo referente a la capacidad actual del dron:

- **capacidad** actual del dron (en paquetes).

> *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. No olvides que también puedes ver la información en el* ***javadoc*** *que adjuntamos a la tarea.*

Además, también se dispondrá de otro 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 drones de reparto** creados hasta ese momento (equivalente al número de objetos **`DronReparto`** instanciados).

Toda la información descrita en este apartado (**y solamente esta**) es la que se debe representar internamente la clase **`DronReparto`**. No se necesitan más atributos. La valoración y decisión de los atributos necesarios para representar nuestros objetos de tipo **`DronReparto`** 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.*

1.3.2.- Constructores

Tendrás que implementar un único **constructor** para la clase ***`DronReparto`***.

1. *Un **constructor con tres parámetros**: **modelo, tipo de motor y capacidad máxima***

Esta clase deberá llamar al constructor de la clase base ***`DispositivoVolador`*** teniendo en cuenta el **modelo** y **tipo de motor**, por lo que el **constructor debe ser de 3 parámetros.**

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, además, la única excepción que debe implementarse en este constructor será:

- si el constructor recibe un un valor negativo de capacidad máxima, la excepción será de tipo **IllegalArgumentException** y el mensaje de error de la excepción debería ser del tipo **"La capacidad máxima no puede ser negativa"**.

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.

1.3.3.- Métodos

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

1. **`getCapacidadMaxima()`**, que devuelva la **capacidad máxima** del dron de reparto (**`int`**);
2. **`getCapacidad()`**, que devuelva la **capacidad actual** del dron de reparto (**`int`**).

*2. Además de esos métodos consultores sobre el estado del objeto, también habrá que implementar algún **método que devuelva información genérica o global sobre la clase (estáticos):***

- **`getNumDronesReparto()`**, que devuelva el **número de drones** de reparto creados hasta el momento (**`int`**).

*3. También tendremos que implementar algunos **métodos para modificar atributos de nuestra clase:***

- **`setCapacidad()`**, que servirá para establecer la capacidad actual del dron. Este método recibirá por parámetro la capacidad actual del dron y no devolverá nada.

*4. También tendremos que redefinir el **método abstracto definido en la clase `DispositivoVolador` para realizar el mantenimiento** de los drones de reparto:*

Para representar el mantenimiento que reciben los drones de reparto sobrescribiremos el siguiente método:

- **`realizarMantenimiento()`**, que servirá para realizar el mantenimiento de los drones de reparto. Este método no tendrá ningún parámetro y no devolverá nada pero mostrará por pantalla el mensaje "**Realizando mantenimiento del dron con modelo xxx: revisión de garras, compartimiento, estabilidad...**", siendo xxx el modelo del dron de reparto.

Con este método comprobaremos cómo en tiempo de ejecución se establece la **ligadura dinámica**.

*5. Debemos sobreescribir el **método `desplazar`** de la clase **`DispositivoVolador.`***

El dron de reparto sobreescribirá el método **`desplazar`** de la clase ***`DispositivoVolador`***. Para poder realizar el desplazamiento del dron de reparto se debe comprobar previamente que éste tenga paquetes cargados. En el caso de que el dron de reparto tenga carga, se realizará un desplazamiento tal y como haría un ***`DispositivoVolador`*** genérico.

Las posibles excepciones que podrían lanzarse son:

- si el dron de reparto no tiene paquetes cargados, la excepción será de tipo **`IllegalStateException`** y el mensaje de error de la excepción debería ser del tipo **"El dron de reparto no se puede desplazar ya que no tiene carga."**.

*6. Además debemos sobreescribir los **métodos** de la interfaz `Repartible`.*

Para poder **cargar paquetes** y posteriormente **repartirlos** habrá que sobreescribir los siguientes métodos que se encargarán de actualizar apropiadamente todos los atributos de estado necesarios, tanto de objeto como de clase, para que se realice adecuadamente el proceso de reparto:

- `cargar()`, que servirá para cargar paquetes en un dron de reparto. Este método recibirá por parámetro el número de paquetes a cargar y no devolverá nada.

En este caso, tendremos que tener en cuenta si alguno de los **parámetros del método es incorrecto** para **lanzar** las correspondientes **excepciones**:

- si se intenta cargar el dron de reparto sin la cantidad mínima de paquetes a cargar, la excepción será de tipo **IllegalArgumentException** y el mensaje de error de la excepción debería ser "**No se puede cargar: El mínimo de paquetes a cargar es 1.**"
- si al intentar cargar el dron de reparto se excede su capacidad máxima, se debería lanzar una excepción de tipo **IllegalStateException** con el mensaje "**No se puede cargar: Se excede la capacidad máxima del dron."**

- `repartir()`, que servirá para repartir los paquetes del dron de reparto. Este método recibirá por parámetro el número de paquetes a repartir y no devolverá nada.

En este caso, también tendremos que tener en cuenta si alguno de los **parámetros del método es incorrecto** para **lanzar** las correspondientes **excepciones**:

- si se intenta repartir menos de un paquete, la excepción será de tipo **IllegalArgumentException** y el mensaje de error de la excepción debería ser "**No se puede repartir: El mínimo de paquetes a repartir es 1.**"
- si se intentan repartir más paquetes de los disponibles, se debería lanzar una excepción de tipo **IllegalStateException** con el mensaje "**No se puede repartir: Se quieren repartir más paquetes de los disponibles."**

*7. Por último, sobrescribiremos el **método `toString` para que devuelva una cadena que represente el estado del dron de reparto en un momento dado.***

Este método `toString` deberá devolver una cadena que represente el **estado del dron de reparto** en un momento dado. Añadirá a siguiente información a la ya devuelta por la clase ***`DispositivoVolador:`***

- la etiqueta ***"Capacidad:"*** junto a la **capacidad actual** del dron de reparto.

```
Modelo=DR-01, Tipo de motor=Combustion, Altitud=0,00 metros, Posicion en X=0, Posicion
                                en Y=0, Capacidad: 10
```

donde observamos que:

- el **modelo** del dron de reparto es **DR-01**;
- el **tipo de motor** del dron de reparto es **Combustion**;
- el dron de reparto se encuentra a una **altitud** de **0,00 metros**;
- el dron de reparto se encuentra en la **posición 0 en el eje X**;
- el dron de reparto se encuentra en la **posición 0 en el eje Y**;
- la **capacidad actual** del dron de reparto es **10**.

1.4.- Clase DronCarreras

Esta clase también hereda de la **superclase** `DispositivoVolador`.

En esta clase definiremos los atributos mínimos para representar las características adicionales que tiene un dron de carreras con respecto a un dispositivo volador genérico.

Esta clase **sí que debe ser instanciable**, ya que sí crearemos objetos de este tipo.

Esta clase **no debe ser heredable**, es decir, ninguna otra clase puede heredar de `DronCarreras`.

JakeysPhotography123 (Pixabay Lincense)

Además, esta clase no implementa ninguna interfaz.

La clase `DronCarreras` debe tener sus **atributos**, **constructores** y **métodos** que describiremos en los siguientes puntos.

1.4.1.- Atributos

Esta clase debe heredar todos sus atributos de la clase **`DispositivoVolador`**, teniendo en cuenta lo siguiente:

> ***Todos los atributos*** *deben ser* ***accesibles únicamente por la clase*** ***`DronCarreras`****, exceptuando las* ***constantes****, que deben ser públicas tal y como se indica más abajo.*

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**:

- **aceleración máxima** del dron de carreras.

> *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.*

Además, también se dispondrá de otro atributo que pertenecerá a la clase (atributos 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 drones de carreras** creados hasta ese momento (equivalente al número de objetos ***`DronCarreras`*** instanciados).

Por último, junto con esos atributos estáticos variables, 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 que utilice la clase:

- el **factor de aceleración** que se utilizará para calcular el consumo del dron de carreras (**1.5**). Atributo **`FACTOR_ACELERACION`**.

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 ***`DronCarreras`***. No se necesitan más atributos. La valoración y decisión de los atributos necesarios para representar nuestros objetos de tipo ***`DronCarreras`*** 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.*

1.4.2.- Constructores

Tendrás que implementar un único **constructor** para la clase `DronCarreras`.

1. *Un **constructor con tres parámetros**: **modelo, tipo de motor y aceleración máxima**.*

Este constructor deberá llamar al constructor de la clase base ***`DispositivoVolador`*** teniendo en cuenta la **aceleración máxima**, por lo que **este constructor debe ser de 3 parámetros.**

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, además, la única excepción que debe implementarse en este constructor será:

- si el constructor recibe un un valor negativo de aceleración máxima, la excepción será de tipo **IllegalArgumentException** y el mensaje de error de la excepción debería ser del tipo **"La aceleración máxima no puede ser negativa"**.

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.

1.4.3.- Métodos

1. Añade a la clase **`DronCarreras`** *el siguiente* ***método de consulta (getter)*** *para poder obtener la siguiente información de objeto:*

- **`getAceleracionMaxima()`**, que devuelva la **aceleración máxima** del dron de carreras (**`double`**).

2. Además, también habrá que implementar algunos ***métodos que devuelvan información genérica o global sobre la clase (estáticos):***

- **`getNumDronesCarreras()`**, que devuelva la cantidad de drones de carreras creados hasta el momento (**`int`**).

3. También tendremos que sobrescribir algún ***método de acción de la clase base*** ***`DispositivoVolador`****.*

Para calcular el consumo del dron de carreras considerando su aceleración máxima reescribiremos el siguiente método:

- **`calcularConsumo()`**, que servirá para calcular el consumo del dron de carreras. Este método recibirá por parámetro la distancia recorrida y devolverá el consumo del dron para dicha distancia recorrida (**`double`**).

En este caso, las posibles excepciones que podrían lanzarse son **las propias que lanza el método definido en la clase base**.

Para calcular el **consumo del dron de carreras** deberemos **añadir** al **consumo** obtenido a través de la llamada al **método definido en la clase base** el **producto** entre la **aceleración máxima** del dron y el **factor de aceleración**.

4. También tendremos que redefinir el ***método abstracto definido en la clase*** ***`DispositivoVolador`*** ***para realizar el mantenimiento de los drones de carreras****:*

Para representar el mantenimiento que reciben los drones de carreras sobrescribiremos el siguiente método:

- **`realizarMantenimiento()`**, que servirá para realizar el mantenimiento de los drones de carreras. Este método no tendrá ningún parámetro y no devolverá nada pero mostrará por pantalla el mensaje "**Realizando mantenimiento del dron con modelo xxx: calibración de hélices, aerodinámica, motor...**", siendo xxx el modelo del dron de carreras.

Con este método comprobaremos cómo en tiempo de ejecución se establece la **ligadura dinámica**.

5. Por último, sobrescribiremos el ***método*** ***`toString`*** ***para que devuelva una cadena que represente el estado del dron de carreras en un momento dado.***

Este método `toString` deberá devolver una cadena que represente el **estado del dron de carreras** en un momento dado. Añadirá a siguiente información a la ya devuelta por la clase ***`DispositivoVolador`***:

- la etiqueta ***"Aceleración máxima: "*** junto a la **aceleración máxima** del dron de carreras.

```
Modelo=DC-01, Tipo de motor=Hibrido, Altitud=0,00 metros, Posicion en X=0, Posicion en Y=0, Aceleración máxima: 100,00
```

donde observamos que:

- el **modelo** del dron de carreras es **DC-01**;
- el **tipo de motor** del dron de carreras es **Hibrido**;
- el dron de carreras se encuentra a una **altitud** de **0,00 metros**;
- el dron de carreras se encuentra en la **posición 0 en el eje X**;
- el dron de carreras se encuentra en la **posición 0 en el eje Y**;
- la **aceleración máxima** del dron de carreras es **100,00**.

1.5.- Documentación javadoc

vargazs (Pixabay Lincense)

Debes documentar apropiadamente el código con **comentarios javadoc** para poder generar una documentación útil y apropiada. Puedes consultar la **guía-resumen** que ya incluimos en los contenidos de la unidad 5 y que replicamos aquí una vez más:

- **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** 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`.

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).**

1.6.- Casos de prueba

En los siguientes subapartados tienes ejemplos de solución de todos los ejercicios propuestos.

1.6.1.- Ejercicio 1

TestEj01

Este ejercicio se comprueban los constructores y atributos para los drones de reparto y de carreras.

```
------------------------------------------------
CASO DE PRUEBAS 01: CONSTRUCTORES Y ATRIBUTOS
------------------------------------------------
-----------------------------------
 - CONSULTA DE ATRIBUTOS PÚBLICOS
-----------------------------------
 -> Factor de aceleración: 1.5

-------------------------------------------------------------------
 - PRUEBA DEL CONSTRUCTOR DE TRES PARÁMETROS (con datos correctos)
-------------------------------------------------------------------
Creando un DronReparto con los parámetros [DR-01,Combustion,5]...
 -> Objeto creado con éxito.

Creando un DronReparto con los parámetros [DR-02,Electrico,10]...
 -> Objeto creado con éxito.

Creando un DronCarreras con los parámetros [DC-01,Combustion,100]...
 -> Objeto creado con éxito.

Creando un DronCarreras con los parámetros [DC-02,Electrico,150]...
 -> Objeto creado con éxito.

-------------------------------------------------------------------
 - PRUEBA DEL CONSTRUCTOR DE TRES PARÁMETROS (con datos no válidos)
-------------------------------------------------------------------
Creando un DronReparto con los parámetros [null,Combustion,5]...
 -> Se ha producido un error: El modelo del dispositivo no puede ser nulo.
Creando un DronReparto con los parámetros [DR-02,,10]...
 -> Se ha producido un error: El tipo de motor no puede estar vacío.
Creando un DronCarreras con los parámetros [,Combustion,10]...
 -> Se ha producido un error: El modelo del dispositivo no puede estar vacío.
Creando un DronCarreras con los parámetros [DC-02,null,10]...
 -> Se ha producido un error: El tipo de motor no puede ser nulo.
```

1.6.2.- Ejercicio 2

TestEj02

Este ejercicio se comprueban los distintos métodos getters y métodos estáticos.

```
------------------------------------------------
CASO DE PRUEBAS 02: GETTERS Y MÉTODOS ESTÁTICOS
------------------------------------------------
------------------------------------------------------------
 CONSULTA DE ATRIBUTOS DE CLASE (antes de crear objetos)
------------------------------------------------------------
 -> Número de drones de reparto creados: 0

 -> Número de drones de carreras creados: 0

---------------------------------
 CREACIÓN DE UN DRONREPARTO DE PRUEBA
---------------------------------
Creando un DronReparto con los parámetros [DR-01,Combustion,5]...
 -> Objeto creado con éxito.

---------------------------------
 CREACIÓN DE UN DRONCARREAS DE PRUEBA
---------------------------------
Creando un DronCarreras con los parámetros [DC-01,Combustion,100]...
 -> Objeto creado con éxito.

---------------------------------
 CONSULTA DE LOS DATOS DEL DRONREPARTO
---------------------------------
Leyendo los datos almacenados en el dron de reparto...
 -> Capacidad máxima del dron de reparto: 5
 -> Capacidad actual del dron de reparto: 0

-----------------------------------
 CONSULTA DE LOS DATOS DEL DRONCARREAS
-----------------------------------
Leyendo los datos almacenados en el dron de carreras...
 -> Aceleración máxima del dron de carreras: 100.0

------------------------------------------------------------
 CONSULTA DE ATRIBUTOS DE CLASE (después de crear objetos)
------------------------------------------------------------
 -> Número de drones de reparto creados: 1
```

```
        -> Número de drones de carreras creados: 1
```

1.6.3.- Ejercicio 3

TestEj03

Este ejercicio se incluyen pruebas con los métodos de acción de los drones de reparto y de carreras.

```
----------------------------------------
CASO DE PRUEBAS 03: MÉTODOS DE ACCIÓN
----------------------------------------
Creando un DronReparto con los parámetros [DR-01,Combustion,5]...
 -> Objeto creado con éxito.

Creando un DronReparto con los parámetros [DR-02,Electrico,10]...
 -> Objeto creado con éxito.

Creando un DronCarreras con los parámetros [DC-01,Combustion,100]...
 -> Objeto creado con éxito.

Creando un DronCarreras con los parámetros [DC-02,Electrico,150]...
 -> Objeto creado con éxito.

--------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (calcular consumo con datos correctos)
--------------------------------------------------------------------------
Calculando consumo para una distancia recorrida de 100...
 -> El DronReparto DR-01 consume 200,00
 -> El DronReparto DR-02 consume 120,00
 -> El DronCarreras DC-01 consume 350,00
 -> El DronCarreras DC-02 consume 345,00

--------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (calcular consumo con datos erróneos)
--------------------------------------------------------------------------
Calculando consumo para una distancia recorrida de -75...
 -> Se ha producido un error: La distancia recorrida no puede ser negativa.

------------------------------------------------------------
 - CONSULTA DE ATRIBUTOS DEL DRONREPARTO (antes de cargar)
------------------------------------------------------------
Leyendo los datos almacenados en el dron de reparto...
 -> Capacidad máxima del dron de reparto: 5
 -> Capacidad actual del dron de reparto: 0

--------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (cargar dron de reparto con datos correctos)
--------------------------------------------------------------------------
```

```
Cargando DronReparto DR-01 con 3 paquetes...
 -> DronReparto cargado con éxito.

------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (cargar dron de reparto con datos incorrectos)
------------------------------------------------------------------------
Cargando DronReparto DR-01 con -2 paquetes...
 -> Se ha producido un error: No se puede cargar: El mínimo de paquetes a cargar es 1
Cargando DronReparto DR-01 con 30 paquetes...
 -> Se ha producido un error: No se puede cargar: Se excede la capacidad máxima del d

------------------------------------------------------------
 - CONSULTA DE ATRIBUTOS DEL DRONREPARTO (después de cargar)
------------------------------------------------------------
Leyendo los datos almacenados en el dron de reparto...
 -> Capacidad máxima del dron de reparto: 5
 -> Capacidad actual del dron de reparto: 3

------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (el dron de reparto reparte con datos correctos)
------------------------------------------------------------------------
El DronReparto DR-01 reparte 2 paquetes...
 -> DronReparto ha repartido con éxito.

------------------------------------------------------------
 - CONSULTA DE ATRIBUTOS DEL DRONREPARTO (después de repartir)
------------------------------------------------------------
Leyendo los datos almacenados en el dron de reparto...
 -> Capacidad máxima del dron de reparto: 5
 -> Capacidad actual del dron de reparto: 1

------------------------------------------------------------------------
 - USO DE LOS MÉTODOS DE ACCIÓN (el dron de reparto reparte con datos incorrectos)
------------------------------------------------------------------------
El DronReparto DR-01 reparte -2 paquetes...
 -> Se ha producido un error: No se puede repartir: El mínimo de paquetes a repartir
El DronReparto DR-01 reparte 4 paquetes...
 -> Se ha producido un error: No se puede repartir: Se quieren repartir más paquetes

----------------------------------------------------------------------
 - PRUEBA DEL MÉTODO ABSTRACTO PARA PROBAR LA LIGADURA DINÁMICA
----------------------------------------------------------------------
-> Realizando el mantenimiento de un DronReparto...
Realizando mantenimiento del dron con modelo DR-01: revisión de garras, compartimento
-> Realizando el mantenimiento de un DronCarreras...
Realizando mantenimiento del dron con modelo DC-01: calibración de hélices, aerodinám
```

1.6.4.- Ejercicio 4

TestEj04

Este ejercicio es para probar el método toString() de los drones de reparto y de carreras.

```
-------------------------------------
CASO DE PRUEBAS 04: MÉTODO toString()
-------------------------------------
Creando un DronReparto con los parámetros [DR-01,Combustion,5]...
 -> Objeto creado con éxito.

Creando un DronCarreras con los parámetros [DC-01,Hibrido,100]...
 -> Objeto creado con éxito.

-----------------------------------------------------
 - PRUEBAS PARA VISUALIZAR EL ESTADO DE LOS OBJETOS
-----------------------------------------------------
Obteniendo los datos del DronReparto...
Modelo=DR-01, Tipo de motor=Combustion, Altitud=0,00 metros, Posicion en X=0, Posicion en Y=0, Capacidad: 0

Obteniendo los datos del DronCarreras...
Modelo=DC-01, Tipo de motor=Hibrido, Altitud=0,00 metros, Posicion en X=0, Posicion en Y=0, Aceleración máxima: 100,00

Cargando DronReparto DR-01 con 4 paquetes...
 -> DronReparto cargado con éxito.

Obteniendo los datos del DronReparto...
Modelo=DR-01, Tipo de motor=Combustion, Altitud=0,00 metros, Posicion en X=0, Posicion en Y=0, Capacidad: 4

Cambiando posición en el eje X del DronReparto a 10...
Cambiando posición en el eje Y del DronReparto a 20...
Cambiando altitud del DronReparto a 30.00...

Cambiando posición en el eje X del DronCarreras a 15...
Cambiando posición en el eje Y del DronCarreras a 5...
Cambiando altitud del DronCarreras a 15.00...

Obteniendo los datos del DronReparto...
Modelo=DR-01, Tipo de motor=Combustion, Altitud=30,00 metros, Posicion en X=10, Posicion en Y=20, Capacidad: 4

Obteniendo los datos del DronCarreras...
Modelo=DC-01, Tipo de motor=Hibrido, Altitud=15,00 metros, Posicion en X=15, Posicion en Y=5, Aceleración máxima: 100,00
```

2.- Información de interés

Recursos necesarios y recomendaciones

mohamed Hassan (Licencia Pixabay)

- Se proporciona un proyecto base sobre el que debes trabajar: Proyecto base (zip - 26 KB). En él tendrás incluidos los paquetes necesarios para trabajar, para que declares e implementes clases e interfaces, así como los programas de pruebas. En este proyecto deberás incluir toda tu jerarquía de clases. **Es obligatorio utilizar este proyecto y sus programas de pruebas** .
- Es fundamental **evitar la redundancia de código** (y de este modo minimizar las posibles inconsistencias), siempre que sea posible, en aquellos fragmentos de código que realicen las mismas acciones, especialmente en los **constructores** y métodos heredados (método `toString`). En esos casos debes aprovechar la llamada desde un método al método homónimo (con el mismo nombre) de la clase padre 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.
- Para que te sea más fácil probar si tu programa funciona correctamente, te proporcionamos también el resultado que debería dar la ejecución de los programas de prueba con los **casos de prueba** que contiene. **Deberías probar al menos que te funcionan correcta y apropiadamente esos casos**, más otros que se te puedan ocurrir a ti, pues el profesorado vamos a probar esos **como mínimo**: Ejecución de programas de prueba
- También te proporcionamos una muestra de cómo podría quedar tu **documentación javadoc**. De esa manera tendrás una descripción precisa y exhaustiva de la interfaz de la clase (nombres de las clases e interfaces, nombres de los atributos públicos y protegidos, nombres de los métodos públicos y protegidos, valores devueltos, parámetros que reciben). Es decir, todo aquello que debe conocer otro programador para poder usar esa biblioteca de clases. La interfaz de tus clases tendrá que ser exactamente igual que lo que se indica en esa documentación (si no, no "encajará" con el programa de prueba). 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): Documentación javadoc (zip - 132 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_PROG06_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

StockSnap (Pixabay Lincense)

Como en la tarea anterior, el proyecto base de esta tarea contiene una serie de **programas de pruebas** (clases `TestEj01`, `TestEj02`, etc.) para probar el funcionamiento de todas las clases paso a paso. Se hacen pruebas de creación de drones de reparto y drones de carreras 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 vuestras clases 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 todas las clases. Es decir, que vuestras clases deberán 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.

2.2.- Excepciones

Al igual que en la tarea anterior, 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 RA7 (*Desarrolla programas aplicando características avanzadas de los lenguajes orientados a objetos y del entorno de programación*):

- a) Se han identificado y evaluado los escenarios de utilización de la herencia y la composición.
- b) Se han identificado los conceptos de herencia, superclase y subclase.
- c) Se han utilizado modificadores para bloquear y forzar la herencia de clases y métodos.
- d) Se ha reconocido la incidencia de los constructores en la herencia.
- e) Se han creado clases heredadas que sobrescriban la implementación de métodos de la superclase.
- f) Se han diseñado y aplicado jerarquías de clases.
- g) Se han probado y depurado las jerarquías de clases.
- h) Se han identificado y evaluado los escenarios de uso de interfaces.
- i) Se han realizado programas que implementen y utilicen jerarquías de clases.
- j) Se ha comentado y documentado el código.

Peggy_Marco (Pixabay License)

¿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				
(RA7.a,RA7.b,RA7.c,RA7.f,RA7.g): Se han utilizado modificadores para implementar la herencia de clases y métodos.	No se han utilizado modificadores para implementar la herencia de clases y métodos. **0 puntos**	Apenas se han utilizado modificadores para implementar la herencia de clases y métodos. **0.5 puntos**	Se han utilizado algunos modificadores para implementar la herencia de clases y métodos. **1 punto**	Se han utilizado bastantes modificadores para implementar la herencia de clases y métodos. **1.5 puntos**	Se han utilizado todos los modificadores para implementar la herencia de clases y métodos. **2 puntos**
(RA7.a,RA7.b,RA7.d,RA7.f,RA7.g): Se han implementado los constructores en la herencia.	No se crean ni emplean los constructores de la clase padre en las subclases. **0 puntos**	Se crean o emplean parcialmente los constructores de la clase padre en las subclases. **0.5 puntos**	Se crean y emplean los constructores de la clase padre en las subclases pero su uso no es del todo correcto. **1 punto**	Se crean y emplean correctamente los constructores de la clase padre en las subclases. **1.5 puntos**	
(RA7.a,RA7.b,RA7.e,RA7.f,RA7.g): Se han creado clases heredadas que sobrescriban la implementación de métodos de la superclase.	No se crean clases heredadas que sobrescriban la implementación de métodos de la superclase. **0 puntos**	Se crean con muchos errores las clases heredadas que sobrescriban la implementación de métodos de la superclase. **0.5 puntos**	Se crean parcialmente las clases heredadas que sobrescriban la implementación de métodos de la superclase. **1 punto**	Se crean todas las clases heredadas que sobrescriban la implementación de métodos de la superclase. pero no son del todo correctas. **1.5 puntos**	Se crean correctamente todas las clases heredadas que sobrescriban la implementación de métodos de la superclase. **2 puntos**
(RA7.a,RA7.b,RA7.f,RA7.g,RA7.h): Se han creado interfaces e implementado sus métodos.	No se implementan las interfaces. **0 puntos**	Apenas se implementan las interfaces y sus métodos y no se utilizan de forma correcta. **0.5 puntos**	Se implementan las interfaces y sus métodos, pero no se utilizan de forma correcta. **1 punto**	Se implementan las interfaces y se utilizan la mayoría de sus métodos de forma correcta. **1.5 puntos**	Se implementan y usan las interfaces junto a sus métodos de forma correcta. **2 puntos**
(RA7.a,RA7.b,RA7.f,RA7.g,RA7.i): Se han realizado programas que implementen y utilicen jerarquías de clases.	No se realizan programas que implementan y utilizan jerarquías de clases. **0 puntos**	Se realizan programas que implementan y utilizan jerarquías de clases con muchos errores. **0.5 puntos**	Se realizan programas que implementan y utilizan jerarquías de clases con algún error. **1 punto**	Se realizan correctamente programas que implementan y utilizan jerarquías de clases. **1.5 puntos**	

(RA7.a,RA7.b,RA7.f,RA7.g,RA7.j): Se ha comentado y documentado el código.	No se comenta ni documenta el código. **0 puntos**	Se incluyen algunos comentarios, pero no se documenta el código. **0.3 puntos**	Se documenta el código, pero los comentarios no son suficientes para entender el código. **0.7 puntos**	Se comenta y documenta apropiadamente el código. **1 punto**

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 las clases no debe romperse**. Si eso sucede es porque alguna 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** que 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 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.

Recordatorio: Si una tarea obtiene calificación cero por alguno de los motivos previamente mencionados, será devuelta al alumno. 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.