

Anexo I.- Ejercicios de implementación de clases

Caso práctico

Una vez que **María y Juan** han comenzado a trabajar en serio con la **Programación Orientada a Objetos**, están empezando a desarrollar clases para algunos de los nuevos proyectos en los que su empresa se encuentra embarcada. Estas clases representarán todo tipo de elementos que puedan resultar de interés para sus aplicaciones.



[Miro Alt \(Pixabay License\)](#)

A continuación se presentan algunos casos prácticos de implementación de clases para que puedas seguir trabajando todo lo que has aprendido durante la unidad.

I.1.- Clase Bombilla

Ejercicio Resuelto

Se nos ha planteado implementar una clase que represente un modelo básico de funcionamiento de una **bombilla**:

- ✓ **estado lumínico de la bombilla (encendida o apagada),**
- ✓ **número de veces que ha sido encendida** desde que se fabricó.

Además de eso se querría mantener un registro de **cuántas bombillas han sido creadas** hasta el momento y **cuántas de ellas se encuentran encendidas**.

Teniendo en cuenta los supuestos anteriores, ¿qué atributos te plantearías incluir en una clase Bombilla?



Ciker-Free-Vector-
Images (Pixabay
License)

[Mostrar retroalimentación](#)

Lo primero que podríamos hacer es considerar qué atributos serían de cada bombilla en particular (atributos de objeto o de instancia) y si habría algún atributo que fuera genérico (de clase).

Para representar el **estado lumínico de un bombilla** en un momento dado (**encendida o apagada**) podría usarse un atributo de tipo **lógico** o **booleano** (boolean en Java), que permite representar dos estados posibles (true y false). Claramente ese atributo sería **de objeto** (cada bombilla tendrá su estado) y **mutable** o variable (el estado puede cambiar a lo largo de la vida del objeto). Un nombre adecuado para este atributo podría ser **estado**.

Para almacenar la **cantidad de veces que una bombilla ha sido encendida**, podría usarse un atributo de tipo **entero** (por ejemplo short). Este atributo también sería de objeto (es una característica de cada bombilla en particular) y **mutable** (cada vez que se encienda se incrementará en uno). Un identificador apropiado para esta característica podría ser, por ejemplo, **vecesEncendida**.

En cuanto al **número de bombillas que han sido creadas**, parece claro que sí se trata de un **atributo de clase** (static en Java), una característica no específica de cada bombilla sino de la clase en general. Para este atributo podría usarse un tipo **entero** (short por ejemplo). Este atributo también será **mutable**, pues irá creciendo a medida que se fabriquen más bombillas (se ejecute algún constructor de la clase). Como nombre para este atributo podríamos escoger, por ejemplo, **bombillasCreadas**.

Por último, para guardar la **cantidad de bombillas que hay encendidas** en cada momento nos encontramos ante otra propiedad no específica de cada bombilla en particular sino genérica de la clase (**atributo de clase**), **mutable** y de tipo **entero** (short). Un identificador adecuado podría ser **bombillasEncendidas**.

Respecto a la **visibilidad**, como suele ser habitual, lo más prudente es que todos estos atributos sean **privados** (private).

A modo de resumen, podríamos construir la siguiente tabla de atributos:

Atributos de la clase Bombilla

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
bombillasCreadas	short	privado (private)	variable	de clase (static)
bombillasEncendidas	short	privado (private)	variable	de clase (static)
<code>estado	boolean	privado (private)	variable	de objeto
vecesEncendida	short	privado (private)	variable	de objeto

A partir de la tabla podemos comenzar a implementar la clase Bombilla en Java con esos atributos:

```
public class Bombilla {

    // Atributos de clase
    private static int bombillasCreadas;
    private static int bombillasEncendidas;

    // Atributos de objeto
    private boolean estado;
    private int vecesEncendida;
}
```

Una vez que tenemos un núcleo básico de atributos para la clase, se nos plantean los siguientes constructores:

- 1.- un **constructor con un solo parámetro** (**estado inicial** de la bombilla), que hará que la bombilla tenga ese estado inicial al crearse;
- 2.- un **constructor sin parámetros**, que hará que la bombilla al crearse esté **apagada**.

Implementa esos constructores en Java y añade, si lo consideras necesario, los atributos adicionales que estimes oportuno.

[Mostrar retroalimentación](#)

Para implementar el primer constructor, habrá que realizar las siguientes acciones:

1. asignar al **estado de la bombilla** el estado inicial que se ha pasado como parámetro. No es necesario llevar a cabo ninguna comprobación para ese parámetro pues sólo podrá tener dos valores (true o false) y ambos son posibles y válidos para el constructor;
2. iniciar el contador que registra el **número de veces que la bombilla ha sido encendida**. Aquí habrá que tener en cuenta si el estado inicial va a ser apagado o encendido, para que este contador comience desde cero o desde uno;
3. actualizar los atributos de clase: **cantidad de bombillas creadas** y **cantidad de bombillas encendidas**.

Dado que no es posible que se produzca ninguna situación anómala o errónea durante la ejecución de este constructor, no se incluirá en su cabecera que lanza (throws) ninguna excepción.

Respecto al **segundo constructor**, podríamos implementarlo fácilmente si hacemos uso de todo lo que hemos llevado a cabo ya en el primero y realizando una simple llamada a this() con el parámetro con valor a false (bombilla apagada), es decir this(false). Ahora bien, si seguimos la política de evitar al máximo el uso de literales en los métodos y usar constantes para que de este modo los literales estén escritos sólo una vez (en la definición de la constante), podríamos utilizar un **atributo de clase (static) constante (final)** para almacenar cuál sería el valor por omisión (o por defecto, del inglés "by default") del estado de una bombilla al crearse. Este atributo, dado que va a ser constante y además refleja cierta información de interés desde fuera de la clase, puede declararse como **público** (public):

```
// Atributos de clase constantes
public static final boolean DEFAULT_ESTADO_INICIAL = false;
```

Teniendo en cuenta todo lo anterior, la implementación de ambos constructores en Java podría quedar así:

```
public Bombilla(boolean estadoInicial) {

    // Inicializamos el estado
    this.estado = estadoInicial;

    // Si el primer estado es encendida, ya ha sido encendida una vez
    this.vecesEncendida = estadoInicial ? 1 : 0;

    // Actualizamos atributos de clase
    Bombilla.bombillasCreadas++;
    Bombilla.bombillasEncendidas += this.estado ? 1 : 0;
}

public Bombilla() {
    this(Bombilla.DEFAULT_ESTADO_INICIAL);
}
```

Analiza e implementa los **getter** (métodos **get**) que consideres apropiado que esta clase podría tener.

[Mostrar retroalimentación](#)

Dados los atributos que tenemos hasta el momento, podríamos definir los siguientes métodos get para obtener **propiedades del objeto** (estado):

- ✓ `getEstado`, para consultar el estado actual de la bombilla (apagada/false o encendida/true);
- ✓ `isEncendida`, que devuelva true si la bombilla está encendida (estado encendido/true);
- ✓ `isApagada`, que devuelva true si la bombilla está apagada (estado apagada/false);
- ✓ `getVecesEncendida`, para obtener el número de veces que ha sido encendida.

```
public boolean getEstado() {
    return this.estado;
}

public boolean isEncendida() {
    return this.estado;
}

public boolean isApagada() {
    return !this.estado;
}

public int getVecesEncendida() {
    return this.vecesEncendida;
}
```

Respecto a la consulta de **propiedades de la clase**, podríamos plantearnos los siguientes métodos:

- ✓ `getBombillasCreadas`, para obtener la **cantidad de bombillas creadas** hasta el momento;
- ✓ `getBombillasEncendidas`, para obtener la **cantidad de bombillas encendidas** en el momento actual.

Es razonable que estos dos getter sean declarados como **métodos de clase o estáticos** (static), pues no hacen uso alguno de los atributos de objeto y por tanto podrían usarse independientemente de la existencia de instancias de la clase Bombilla.

```
public static int getBombillasCreadas() {
    return Bombilla.bombillasCreadas;
}

public static int getBombillasEncendidas() {
    return Bombilla.bombillasEncendidas;
}
```

Observa cómo al consultar **atributos de objeto** hemos utilizado la referencia `this` y sin embargo para acceder a **atributos de clase** hemos recurrido a la referencia de la propia clase (`Bombilla`).

Una vez que ya disponemos de constructores y métodos de consulta de atributos, es el momento de plantearse un posible método `toString` para esta clase. Nos han propuesto el siguiente formato de salida:

`Bombilla xxx. Se ha encendido zzz veces`

donde `xxx` será el estado de la bombilla (encendida o apagada) y `zzz` el número de veces que ha sido encendida hasta el momento. También queremos que si la bombilla se ha encendido una sola vez, aparezca la palabra "vez" (singular) y no "veces" (plural). Por ejemplo:

```
Bombilla apagada. Se ha encendido 0 veces
Bombilla encendida. Se ha encendido 1 vez
Bombilla apagada. Se ha encendido 3 veces
```

Teniendo en cuenta esas especificaciones, implementa un método `toString` en Java para la clase `Bombilla`.

[Mostrar retroalimentación](#)

Teniendo en cuenta las especificaciones anteriores, el método `toString` podría quedar así:

```
@Override
public String toString() {
    String resultado = String.format("Bombilla %s. Se ha encendido %d %s",
        this.estado ? "encendida" : "apagada",
        this.vecesEncendida,
        this.vecesEncendida == 1 ? "vez" : "veces");

    return resultado;
}
```

Si te fijas, se ha incluido la anotación `@Override` antes de la cabecera del método para indicar que se trata de un método "sobrescrito" o "redefinido" (`override`). Esto es porque toda clase en Java ya dispone de ese método `toString` heredado de la clase `Object` y lo que hemos hecho ha sido volverlo a escribir para adaptarlo a nuestras necesidades. Esto lo comprenderás mejor cuando trabajes más adelante con la **herencia**.

Respecto a los posibles **métodos de acción** que se pueden aplicar sobre un objeto bombilla se ha pensado en los siguientes:

- 1.- método para **encender** una bombilla;
- 2.- método para **apagar** una bombilla;
- 3.- método para **comutar** el estado de una bombilla (pasarla de encendida a apagada o viceversa).

Debes tener en cuenta que **una bombilla sólo puede ser encendida si aún no lo está**. Si se intenta encender una bombilla encendida que ya lo está, debería producirse algún tipo de señal que indicara que se ha intentado llevar a cabo una acción que no se podía o no se debía realizar en ese momento. ¿Qué mecanismo proporciona Java para informar sobre ese tipo de circunstancias? También debería tenerse en cuenta lo mismo para cuando intente apagarse una bombilla que ya se encuentra apagada.

También **debes reflexionar acerca de todos los efectos que producen estas acciones sobre todos y cada uno de los atributos de objeto y de clase**, no sólo de los más evidentes. Está claro que encender o apagar una bombilla afectará al estado lumínico de la bombilla (apagada o encendida), pero es probable que afecte también a otros atributos (número de veces que se ha encendido la bombilla, cantidad de bombillas encendidas, etc.). Ese análisis habrá que realizarlo siempre que se vaya a implementar un nuevo método.

Teniendo en cuenta todas las consideraciones anteriores, implementa esos tres métodos para la clase `Bombilla`.

[Mostrar retroalimentación](#)

Los tres métodos planteados se basan básicamente en el cambio de estado de encendido a apagado o viceversa.

Analicemos cada uno de los métodos, observando:

- 1.- qué circunstancias hay que comprobar antes de llevar a cabo ninguna modificación;
- 2.- qué errores pueden producirse y cómo podemos informar de ellos;
- 3.- a qué atributos afectaría la ejecución de esa acción.

Para el caso del método para **encender** una bombilla (método encender):

- 1.- si la bombilla ya está encendida, no hay que hacer nada, si acaso enviar una señal de que la bombilla no puede encenderse porque ya lo está;
- 2.- podríamos informar de que se intenta encender una bombilla ya encendida mediante una excepción de tipo **IllegalStateException**;
- 3.- si finalmente hay que encender la bombilla porque en efecto estaba apagada, los atributos afectados serían:
 - 3.1.- el **estado de la bombilla** (atributo estado), que pasaría de apagado (false) a encendido (true);
 - 3.2.- el **número de veces que la bombilla ha sido encendida** (atributo vecesEncendida), que habría que incrementarlo en uno;
 - 3.3.- el **número de bombillas encendidas en ese momento** (atributo de clase bombillasEncendida), que habría que incrementarlo también en uno.

Teniendo en cuenta todo eso, la implementación del método encender en Java podría quedar así:

```
public void encender() throws IllegalStateException {
    if ( !this.estado ) { // Comprobamos que la bombilla no esté aún encendida
        this.estado = true; // Pasamos el estado a encendida
        this.vecesEncendida++; // Incrementamos el número de veces que la bombilla ha sido encendida
        Bombilla.bombillasEncendidas++; // Incrementamos la cantidad de bombillas encendidas en este momento
    } else { // Si la bombilla ya está encendida se lanza una excepción
        throw new IllegalStateException ("intentando encender bombilla ya encendida");
    }
}
```

Dado que este método no necesita devolver ningún valor, será de tipo void.

Respecto a la acción de **apagar** una bombilla (método apagar), será similar, teniendo en cuenta que hace prácticamente lo contrario. En este caso los atributos afectados serían únicamente estado y **bombillasEncendidas**.

```
public void apagar () throws IllegalStateException {
    if ( !this.estado ) { // Si la bombilla ya está apagada se lanza una excepción
        throw new IllegalStateException ("Intentando apagar bombilla apagada");
    } else { // Sólo si la bombilla está encendida se llevan a cabo las acciones
        this.estado= false; // Pasamos el estado a apagada
        Bombilla.bombillasEncendidas--; // Decrementamos en uno el número de bombillas encendidas en este momento
    }
}
```

En cuanto a la acción de **comutar** el estado de una bombilla (método comutar), no podrá producirse ningún error, pues se pasará al estado contrario en el que se encuentre la bombilla: `this.estado= !this.estado`.

Aunque claro, como hacemos siempre, hay que tener en cuenta también el resto de atributos que pueden verse afectados:

```
public void comutar() {
    this.estado= !this.estado; // Cambiamos al contrario o "comutamos" el valor del estado
    if (this.estado) { // Si la hemos encendido (estado es true)
        this.vecesEncendida++; // Incrementamos el número de veces que se ha encendido
        Bombilla.bombillasEncendidas++; // Incrementamos la cantidad de bombillas encendidas
    } else { // Si la hemos apagado (estado es false)
        Bombilla.bombillasEncendidas--; // Decrementamos la cantidad de bombillas encendidas
    }
}
```

Esa sería una manera de plantearse el método implementándolo desde cero. Ahora bien, podríamos aprovechar todo lo que ya había implementado (métodos encender y apagar) de manera que les dejaríamos a ellos el trabajo "duro" de actualizar en cada caso los atributos afectados:

```
public void comutar() {
    if (this.estado)
        apagar(); // Si la bombilla está encendida, la apagamos
    else
        encender(); // Si la bombilla está apagada, la encendemos
}
```

Implementar el método comutar de esta manera tiene la ventaja de, además de ahorrar código (que en realidad tampoco ahorraremos tanto), separar el trabajo de cada método y responsabilizar a cada uno de una función específica y diferente (sin mezclar acciones comunes). Así, de esta manera, dejamos al método apagar la responsabilidad de actualizar adecuadamente todos los atributos que haya que modificar cuando se apague una bombilla (y lo mismo para el método encender). Como consecuencia, el método comutar se sirve de ellos y no vuelve a implementar esas acciones.

Si en el futuro se ampliara la clase Bombilla para que contuviera y gestionara más atributos, es probable que el método conmutar no hubiera que modificarlo, pues la actualización de atributos se realizaría en los métodos apagar y encender y no aquí. De esa manera estaríamos implementando un **código mucho más independiente y fácil de gestionar**.

Ejercicio Resuelto

Una vez que disponemos de la clase Bombilla, la idea es poder utilizarla en tus programas, llevando a cabo acciones como:

- ✓ **declaración de variables** referencia a esa clase,
- ✓ **creación de objetos** instancia esa clase mediante la invocación a **constructores**,
- ✓ **manipulación** de esos objetos a través de sus métodos,
- ✓ visualización por la consola de los atributos de los objetos mediante el uso de getters o del método `toString`,
- ✓ etc.



[Bru-nQ \(Pixabay License\)](#)

Supongamos que queremos implementar un programa en Java que use objetos de la clase Bombilla para llevar a cabo lo siguiente:

- 1.- se muestre la **cantidad de bombillas creadas** hasta el momento mediante el uso de método `getBombillasCreadas`;
- 2.- se muestre la **cantidad de bombillas encendidas** en este momento mediante el uso de método `getBombillasEncendidas`;
- 3.- se declaren dos variables `b1`, `b2` y `b3` de tipo referencia a la clase Bombilla;
- 4.- se **instancie una primera bombilla**, se asigne su referencia a la variable `b1` usando el **constructor sin parámetros** y se muestre su estado usando el método `getEstado`;
- 5.- se **instancie una segunda bombilla encendida**, se asigne su referencia a la variable `b2` usando el **constructor con parámetros** y se muestre su estado usando el método `toString`;
- 6.- para esta segunda bombilla , que se realicen las siguientes acciones (tras cada acción se mostrará su estado mediante el método `toString`):
 - 6.1.- se **conmuta** su estado **tres veces seguidas** mediante un bucle for;
 - 6.2.- se **enciende** dos veces seguidas;
 - 6.3.- se **apaga** una vez;
 - 6.4.- se **vuelve a encender**;
- 7.- se **instancie una tercera bombilla encendida**, se asigne su referencia a la variable `b3` usando el **constructor con parámetros** y se muestre su estado usando el método `toString`;
- 8.- se muestre la **cantidad de bombillas creadas** hasta este momento mediante el uso de método `getBombillasCreadas`;
- 9.- se muestre la **cantidad de bombillas encendidas** en este momento mediante el uso de método `getBombillasEncendidas`.

¿Qué salida por pantalla deberíamos obtener?

[Mostrar retroalimentación](#)

Deberíamos obtener una salida del siguiente tipo:

Número de bombillas creadas hasta el momento: 0.
Número de bombillas encendidas en este momento: 0.

Declarando variables `b1`, `b2`, `b3` de tipo referencia a objetos Bombilla.

Creación de `b1`.
Estado de `b1`: false

Creación de `b2`.
Estado de `b2`: Bombilla encendida. Se ha encendido 1 vez.

Comutando estado de `b2`.
Estado de `b2`: Bombilla apagada. Se ha encendido 1 vez.

Comutando estado de `b2`.
Estado de `b2`: Bombilla encendida. Se ha encendido 2 veces.

Comutando estado de `b2`.
Estado de `b2`: Bombilla apagada. Se ha encendido 2 veces.

Encendiendo `b2`.
Estado de `b2`: Bombilla encendida. Se ha encendido 3 veces.

Apagando `b2`.
Estado de `b2`: Bombilla apagada. Se ha encendido 3 veces.

Encendiendo b2.

Estado de b2: Bombilla encendida. Se ha encendido 4 veces.

Creación de b3.

Estado de b3: Bombilla encendida. Se ha encendido 1 vez.

Número de bombillas creadas hasta el momento: 3.

Número de bombillas encendidas en este momento: 2.

Implementa una clase de pruebas para la clase Bombilla que contenga un método main donde se lleven a cabo las acciones anteriores.

[Mostrar retroalimentación](#)

El código del método main de la clase de pruebas podría quedar más o menos así:

```
public static void main(String[] args) {

    // Valores de clase iniciales
    System.out.printf ("Número de bombillas creadas hasta el momento: %d.\n", Bombilla.getBombillasCreadas());
    System.out.printf ("Número de bombillas encendidas en este momento: %d.\n\n", Bombilla.getBombillasEncendidas());

    // Declaración de variables
    System.out.printf ("Declarando variables b1, b2, b3 de tipo referencia a objetos Bombilla.\n\n");
    Bombilla b1, b2, b3;

    // Creación y asignación del primer objeto
    System.out.printf ("Creación de b1.\n");
    b1= new Bombilla();
    System.out.printf ("Estado de b1: %s\n\n", b1.getEstado());

    // Creación y asignación del segundo objeto
    System.out.printf ("Creación de b2.\n");
    b2= new Bombilla(true);
    System.out.printf ("Estado de b2: %s.\n\n", b2);

    // Comutar b2 cuatro veces
    for (int i=0; i < 4; i++) {
        System.out.printf ("Comutando estado de b2.\n");
        b2.comutar();
        System.out.printf ("Estado de b2: %s.\n\n", b2);
    }

    // Encender b2
    try {
        System.out.printf ("Encendiendo b2.\n");
        b2.encender();
    } catch (IllegalStateException ex) {
        System.out.printf ("Error: %s.\n", ex.getMessage());
    } finally {
        System.out.printf ("Estado de b2: %s.\n\n", b2);
    }

    // Encender b2 (error)
    try {
        System.out.printf ("Encendiendo b2.\n");
        b2.encender();
    } catch (IllegalStateException ex) {
        System.out.printf ("Error: %s.\n", ex.getMessage());
    } finally {
        System.out.printf ("Estado de b2: %s.\n\n", b2);
    }

    // Apagar b2
    try {
        System.out.printf ("Apagando b2.\n");
        b2.apagar();
    } catch (IllegalStateException ex) {
        System.out.printf ("Error: %s.\n", ex.getMessage());
    } finally {
        System.out.printf ("Estado de b2: %s.\n\n", b2);
    }

    // Encender b2
}
```

```

try {
    System.out.printf ("Encendiendo b2.\n");
    b2.encender();
} catch (IllegalStateException ex) {
    System.out.printf ("Error: %s.\n", ex.getMessage());
} finally {
    System.out.printf ("Estado de b2: %s.\n\n", b2);
}

// Creación y asignación del tercer objeto
System.out.printf ("Creación de b3.\n");
b3= new Bombilla(true);
System.out.printf ("Estado de b3: %s.\n\n", b3);

// Valores de clase finales
System.out.printf ("Número de bombillas creadas hasta el momento: %d.\n", Bombilla.getBombillasCreadas());
System.out.printf ("Número de bombillas encendidas en este momento: %d.\n", Bombilla.getBombillasEncendidas());
}

```

¿Podrías haber implementado esas pruebas dentro de la propia clase Bombilla? ¿Cómo podrías haberlo hecho?

[Mostrar retroalimentación](#)

Sí. Podrías haber introducido todo ese código de prueba dentro de un método main dentro de la clase Bombilla en lugar de hacer otra clase independiente para llevar a cabo las pruebas.

Lo habitual es hacer las pruebas en un método main en una clase independiente a la clase que quieras probar para no "ensuciar" la clase con un método main que contiene unas pruebas que tienen una función muy específica y que luego no se van a utilizar para nada, pero es bueno que sepas que puede hacerse por si en alguna ocasión te encuentras algo así.

Ejercicio Resuelto

Una vez que ya disponemos de una implementación básica de la clase Bombilla, los analistas de nuestra empresa han venido con unas nuevas especificaciones para el equipo de desarrollo. Nos piden ahora que los objetos de la clase Bombilla también sean capaces de calcular y registrar el consumo de cada bombilla.

Los aparatos eléctricos cuando están funcionando generan un consumo de energía eléctrica en función de la potencia que tengan y del tiempo que estén en funcionamiento. Por tanto, para calcular el **consumo** producido por un dispositivo eléctrico es necesario conocer la **potencia** de ese aparato y el **tiempo** que ha estado funcionando. La potencia en el S.I. se mide en vatios (símbolo **W**).

El consumo de un dispositivo eléctrico se calcula multiplicando su potencia por la cantidad de tiempo que ha sido usado ese dispositivo. Es decir:

$$\text{consumo (energía consumida)} = \text{potencia} \times \text{tiempo}$$

donde el consumo se mediría en **kilovatios-hora (kWh)**, la potencia en **vatios (W)** y el tiempo en **horas**.

Ante estos nuevos requerimientos, hay una serie de preguntas que puedes plantearte:

- ✓ ¿qué **nuevos atributos** vamos a necesitar para poder tener en cuenta estas nuevas circunstancias? ¿Habrá nuevos atributos de objeto o de clase? ¿De qué tipo serán? ¿Serán constantes o variables?
- ✓ ¿cómo va a afectar a los **constructores** y **métodos ya existentes**? ¿Habrá que añadirles nuevos parámetros? ¿Será necesario llevar a cabo más comprobaciones? ¿Cómo afectará a la actualización de los atributos ya existentes? ¿Y a la de los nuevos atributos?
- ✓ ¿será necesario implementar **nuevos métodos**? ¿Con qué parámetros? ¿Con qué posibles excepciones?

Teniendo en cuenta todas estas nuevas cuestiones, replantea el nuevo escenario de **atributos** de la nueva clase Bombilla.

[Mostrar retroalimentación](#)

En este nuevo escenario de atributos está claro que aparecen al menos dos nuevos actores:

- ✓ la **potencia** de la bombilla (en **W**), que dado que se trata de una magnitud física será de tipo **real** (por ejemplo **double**), **inmutable** (final), pues una vez creada una bombilla su potencia en principio no debería cambiar), **de objeto** (cada bombilla tendrá una potencia determinada) y **privada** (private). A este atributo lo podríamos llamar potencia;
- ✓ el **tiempo** que la bombilla ha estado encendida desde que se creó hasta que se apagó por última vez (en horas). Para ello podríamos tener un atributo de tipo **real** (**double**), **mutable** (cuanto más tiempo pase estando encendida, más se incrementará este atributo), **de objeto** (cada bombilla llevará una determinada cantidad de tiempo encendida) y **privada** (private). A ese atributo lo podríamos llamar tiempoEncendida.



A. Meškauskas (CC BY-SA)

Si una bombilla está apagada, está claro que el tiempo que ha estado encendida estará registrado en el atributo `tiempoEncendida` (todo el tiempo acumulado de todas las veces que ha estado encendida), pero si está encendida habría que sumar a ese tiempo el tiempo que ha transcurrido desde que se encendió por última vez hasta el instante actual (en el que aún permanece encendida). Eso nos indica que necesitamos también almacenar en un atributo el instante (fecha y hora) en el que se encendió por última vez. Se trataría de un atributo que también sería **privado**, **variable** (o mutable), y **de objeto**. Un tipo adecuado para almacenar esa información en Java podría ser un objeto de la clase `LocalDateTime`. Un posible nombre para ese atributo podría ser: `ultimaVezEncendida`.

Respecto a la **potencia** de una bombilla, está claro que se trata de una característica intrínseca del objeto (**atributo de objeto inmutable**) y que habría que proporcionarla a la hora de crear un nuevo objeto de tipo Bombilla, es decir, como **parámetro en el constructor**. Para evitar que se reciban valores inválidos o inadecuados para ese parámetro (potencia negativa, potencias de trillones de vatios, etc.), debería realizarse una comprobación de rango para asegurarse de que se trata de una potencia válida. Como ya hemos hecho otras veces, sería razonable definir esos rangos (valores literales) como atributos constantes para evitar el uso de literales en los métodos. Para este caso podríamos definir:

- ✓ **Máxima potencia** permitida para una bombilla, por ejemplo 200 W.
- ✓ **Minima potencia** permitida para una bombilla, por ejemplo 10 W.
- ✓ **Potencia por omisión** para una bombilla, por ejemplo 60 W. Utilizada por si implementamos un constructor al que no se le pasa la potencia como parámetro.

Podríamos llamar a estos atributos `MAXIMA_POTENCIA`, `MINIMA_POTENCIA` y `DEFAULT_POTENCIA`.

Teniendo en cuenta todo este análisis, la nueva tabla de atributos de nuestra clase Bombilla podría quedar de la siguiente manera:

Atributos de la nueva clase Bombilla

Nombre	Tipo	Visibilidad	Mutabilidad	De objeto/de clase
DEFAULT_ESTADO_INICIAL	boolean	público (public)	constante (final)	de clase (static)
MAXIMA_POTENCIA	double	público (public)	constante (final)	de clase (static)
MINIMA_POTENCIA	double	público (public)	constante (final)	de clase (static)
DEFAULT_POTENCIA	double	público (public)	constante (final)	de clase (static)
bombillasCreadas	short	privado (private)	variable	de clase (static)
bombillasEncendidas	short	privado (private)	variable	de clase (static)
estado	boolean	privado (private)	variable	de objeto
vecesEncendida	short	privado (private)	variable	de objeto
potencia	double	privado (private)	constante (final)	de objeto
tiempoEncendida	double	privado (private)	variable	de objeto
ultimaVezEncendida	LocalDateTime	privado (private)	variable	de objeto

A partir de la tabla podemos comenzar a implementar la nueva clase Bombilla en Java con todos esos atributos:

```
public class Bombilla {

    // ATRIBUTOS DE CLASE
    // -------

    // Atributos constantes de clase
    public static final boolean DEFAULT_ESTADO_INICIAL = false;

    public static final double MAXIMA_POTENCIA= 200.00; // (en vatios)
    public static final double MINIMA_POTENCIA= 10.00; // (en vatios)
    public static final double DEFAULT_POTENCIA= 60.00; // (en vatios)

    // Atributos variables de clase
    private static short bombillasCreadas;
    private static short bombillasEncendidas;

    // ATRIBUTOS DE OBJETO
    // -------

    // Atributos constantes de objeto
    private final double potencia; // (en vatios)

    // Atributos de variables objeto
    private boolean estado;
    private int vecesEncendida;
    private double tiempoEncendida; // (en horas)
    private LocalDateTime ultimaVezEncendida;

}
```

Una vez que ya tenemos los nuevos atributos, habrá que plantearse cómo afecta eso a los constructores que ya teníamos y si podría venir bien disponer de algún otro constructor adicional.

Reestructura los constructores que ya tenías para la clase Bombilla y, si lo consideras apropiado, añade alguno nuevo que pienses que podría resultar útil.

[Mostrar retroalimentación](#)

Para empezar, podríamos añadir a los constructores ya existentes un nuevo parámetro que indique la potencia de la bombilla. De esta manera, el constructor con un parámetro (**estado inicial**) pasaría a tener **dos parámetros** (**estado inicial y potencia**):

```
public Bombilla (boolean estadoInicial, double potencia) {

    // Comprobamos si la potencia es válida
    if (potencia < Bombilla.MINIMA_POTENCIA || potencia > Bombilla.MAXIMA_POTENCIA) {
        throw new IllegalArgumentException(
            String.format("potencia no válida: %.2f", potencia));
    }

    // Inicializamos los atributos intrínsecos
    this.potencia= potencia;

    // Inicializamos los atributos de estado
    this.estado = estadoInicial;

    // Si el primer estado es encendida, ya ha sido encendida una vez
    this.vecesEncendida = estadoInicial ? 1 : 0;

    // Si la bombilla comienza como encendida, registramos el instante
    // actual como último instante de encendido
    if (this.estado) {
        this.ultimaVezEncendida= LocalDateTime.now();
    } else {
        this.ultimaVezEncendida= null;
    }

    // Actualizamos atributos de clase
    Bombilla.bombillasCreadas++;
    Bombilla.bombillasEncendidas += this.estado ? 1 : 0;
}
```

El **constructor sin parámetros** podría pasar a tener un parámetro (**potencia**):

```
public Bombilla (double potencia) throws IllegalArgumentException {
    this(Bombilla.DEFAULT_ESTADO_INICIAL, potencia);
}
```

Y podríamos dejar el antiguo constructor de un parámetro con el **estado inicial** (suponiendo una potencia por omisión):

```
public Bombilla (boolean estadoInicial) { // No lanza excepciones porque no puede haber errores
    this(estadoInicial, Bombilla.DEFAULT_POTENCIA);
}
```

Y por último podríamos también implementar **un nuevo constructor sin parámetros** que suponga un **estado inicial por omisión** y una **potencia por omisión**. Para hacerlo podríamos llamar a uno de los constructores con un parámetro y usando el otro valor por omisión o bien invocando al constructor con dos parámetros usando los dos valores por omisión. Nosotros hemos escogido la primera opción:

```
public Bombilla () { // No lanza excepciones porque no puede haber errores
    this(Bombilla.DEFAULT_ESTADO_INICIAL);
}
```

La elección de unas posibilidades u otras a la hora de implementar constructores dependerá de las especificaciones de diseño que nos hayan proporcionado o bien, si no nos han dado ninguna directriz, de aquellos constructores que veas más útiles y apropiados a la hora de trabajar con esta clase. En nuestro caso hemos decidido implementar **cuatro constructores**.

¿Y si nos hubiéramos planteado implementar también un **constructor copia**?

Lo primero sería decidir qué atributos deseamos replicar en la copia: ¿sólo los "intrínsecos" (constantes) o también los de estado? Si elegimos la primera opción, el constructor de copia podríamos implementarlo llamando al constructor de un parámetro que recibe la potencia de la bombilla (que copiaremos de la bombilla original). El resto de atributos se inicializarán como si fuera una bombilla recién creada (sin consumo, cero veces encendida, etc.):

```
public Bombilla (Bombilla bombillaOriginal) throws NullPointerException {
    this (bombillaOriginal.potencia);
}
```

Analiza e implementa los nuevos **getter** (métodos **get**) que podría tener la clase Bombilla.

[Mostrar retroalimentación](#)

Dados los nuevos atributos de la clase, podríamos inicialmente plantear un getter por cada uno de ellos:

- ✓ **potencia** de la bombilla;
- ✓ **tiempo acumulado en que ha estado encendida** la bombilla hasta que se apagó por última vez;
- ✓ **último instante en el que fue encendida** la bombilla.

Ahora bien, quizá algunos de estos atributos no proporcionen una información completa y útil para el usuario del objeto.

En el caso de la **potencia**, es una característica de la bombilla que parece apropiado obtener con un getter "tradicional" (devolver simplemente el valor de un atributo: método `getPotencia`). En el caso de los otros dos atributos se trata de información parcial que podría resultar confusa o poco útil para el usuario del objeto, pero que podría combinarse para obtener información algo más elaborada:

- ✓ **tiempo total en que ha estado encendida** la bombilla desde que se creó (el acumulado más el que ha pasado desde que se encendió por última vez hasta el instante actual). Para ello habrá que usar el tiempo acumulado el último instante de encendido y la fecha y hora actual;
- ✓ **consumo total** desde que se creó la bombilla.

La implementación del método `getPotencia` no tiene ninguna dificultad:

```
public double getPotencia () {
    return this.potencia;
}
```

Para calcular el **tiempo total acumulado que lleva la bombilla encendida** habrá que tener en cuenta dos factores:

- 1.- el **tiempo acumulado en que ha estado encendida** la bombilla hasta que se apagó por última vez;
- 2.- si la bombilla está actualmente encendida: el **tiempo transcurrido desde que se encendió por última vez hasta el instante actual**. Si la bombilla está actualmente apagada, este segundo factor sería cero.

El método que lleve a cabo ese cálculo podríamos considerarlo un getter "indirecto" (llamándolo por ejemplo `getTiempoEncendida`) o bien darle otro nombre (por ejemplo `calcularTiempoEncendida`). En cualquier caso la implementación de ese algoritmo podría quedar así:

```
private double calcularTiempoEncendida () {
    double primerTramo= this.tiempoEncendida;
    double segundoTramo= 0.0;

    // Calculamos el segundo tramo de tiempo encendida
    // (desde que se encendió por última vez hasta el instante actual)
    if (this.ultimaVezEncendida != null) {
        segundoTramo= this.ultimaVezEncendida.until (
            LocalDateTime.now(), ChronoUnit.SECONDS) / 3600.0; // cálculo en horas
    }

    // Sumamos el primer tramo y el segundo tramo de tiempo encendida
    return primerTramo + segundoTramo;
}
```

Este método podría ser privado y usarse internamente como herramienta para el cálculo del consumo, o bien dejarlo como público para que pueda usarlo todo usuario de la clase. Si lo dejamos como público quizás podría usarse el nombre de `getTiempoEncendida` en lugar de `calcularTiempoEncendida`. Será una cuestión que tendrán decidir los desarrolladores de la clase:

- ✓ opción 1: `private double calcularTiempoEncendida ()`
- ✓ opción 2: `public double getTiempoEncendida ()`

Respecto al método que calcula el **consumo total de la bombilla**, una vez implementado el método anterior, sería muy sencillo pues simplemente habría que multiplicar por la potencia:

```
public double getPotenciaConsumida () {
    return calcularTiempoEncendida() * this.potencia;
}
```

En este caso sucede algo parecido respecto al nombre: `calcularPotenciaConsumida` o `getPotenciaConsumida`, según se considere.

Respecto a los **métodos de acción** de los que ya disponíamos:

- 1.- método para **encender** una bombilla;
- 2.- método para **apagar** una bombilla;
- 3.- método para **comutar** el estado de una bombilla;

analiza qué habría que incluir en sus implementaciones para tener en cuenta las nuevas especificaciones de la clase.

[Mostrar retroalimentación](#)

Cuando una bombilla se enciende, hay que **registrar el instante en que es encendida** (atributo `ultimaVezEncendida`) y **cuando se apaga** hay que **recalcular el tiempo acumulado que ha estado encendida** (atributo `tiempoEncendida`).

Teniendo esto en cuenta la implementación de los métodos `encender` y `apagar` podría quedar así:

```
public void encender() throws IllegalStateException {  
    if ( !this.estado ) {  
        this.estado = true;  
        this.vecesEncendida++;  
        this.ultimaVezEncendida= LocalDateTime.now(); // Registrar el momento actual  
    } else {  
        throw new IllegalStateException ("Intentando encender bombilla encendida");  
    }  
}  
  
public void apagar () throws IllegalStateException {  
    if ( !this.estado ) {  
        throw new IllegalStateException ("Intentando apagar bombilla apagada");  
    } else {  
        this.estado= false;  
        this.tiempoEncendida += this.ultimaVezEncendida.until(LocalDateTime.now(), ChronoUnit.SECONDS) / 3600; // Recacular  
        this.ultimaVezEncendida= null; // Ya no está encendida  
    }  
}
```

Respecto al método `comutar`, si hemos implementado apropiadamente los métodos `encender` y `apagar` no haría falta hacer ninguna modificación en él.

I.2.- Clase Ticket

Ejercicio Resuelto

Nuestro equipo de desarrollo ha recibido instrucciones para implementar una clase Ticket cuyos objetos representen un ticket de entrada a un servicio. Se han establecido los siguientes requerimientos:

- ✓ todo ticket debe incluir un identificador que tendrá el siguiente formato: "NNNN-XXXXXXX" donde "NNNN" representa las cuatro cifras del año en el que se ha generado el ticket y "XXXXXXX" un número de ocho cifras que empezará en "00000001" para cada año;
- ✓ cada vez que se genere un nuevo ticket ese código avanza en un número y cada vez que se cambie de año habrá que comenzar de nuevo por "00000001". Esto significa que el primer ticket que se genere en el año 2021 tendrá el identificador "2021-00000001", el segundo "2021-00000002" y así sucesivamente;
- ✓ cuando el último ticket generado tenga el número de secuencia "99999999", ya no se podrán generar más tickets para ese año;
- ✓ todo ticket tendrá asociada una fecha (un día, mes y año) durante el cual podrá ser utilizado. Fuera de esa fecha no podrá usarse;
- ✓ los tickets sólo pueden usarse una vez. Una vez usados quedan marcados con la hora en la que han sido utilizados y no podrán volver a usarse.



[sbqonti \(Pixabay License\)](#)

Realiza un análisis inicial para decidir qué atributos necesitarías para implementar apropiadamente esta clase.

[Mostrar retroalimentación](#)

Necesitaríamos almacenar la siguiente información:

- ✓ por cada ticket (atributos de objeto):
 - ↳ la fecha del ticket, **inmutable**. Podría usarse un objeto de la clase LocalDate.
 - ↳ el identificador (cadena de tipo "NNNN-XXXXXXX"), **inmutable**, de tipo String.
 - ↳ si el ticket ha sido usado o no, **mutable**. Podríamos usar un boolean.
 - ↳ la hora a la que ha sido usado el ticket. **Mutable**. Podríamos usar un LocalTime o un String. Podríamos usar este mismo atributo para indicar si el ticket ha sido usado o no (si tiene el valor null es que aún no ha sido usado y si contiene una hora, es que ya se ha utilizado). Así nos ahorraríamos el atributo anterior.
- ✓ para todos los tickets (atributos de clase):
 - ↳ el último número de secuencia utilizado para así poder generar el siguiente. **Mutable**, de tipo entero (por ejemplo int).
 - ↳ el año del último ticket que se ha generado, para saber si se ha cambiado de año y hay que reiniciar la secuencia de identificadores. **Mutable**, de tipo entero (por ejemplo short, pues es para almacenar un año).
 - ↳ el máximo número de secuencia para el identificador de ticket (valor 99999999). Así evitamos usar literales en los métodos. Este tipo de atributos suele ser público, pero dado que en este caso el usuario de la clase no puede decidir el número de secuencia, no se trata de una información que le pueda ayudar a evitar errores, así que podríamos dejarlo como **privado** en esta ocasión.

A partir de esas decisiones, los atributos de la clase Ticket podrían quedar así:

```
public class Ticket {

    // ATRIBUTOS DE CLASE
    // -----
    // Atributos de clase constantes
    private static int MAX_SEQUENCE = 99_999_999; // Máximo número de secuencia en id

    // Atributos de clase variables
    private static short lastYear = (short) LocalDate.now().getYear(); // Año actual
    private static int lastSequence = 0; // Inicio de secuencia

    // ATRIBUTOS DE OBJETO
    // -----
    // Atributos de objeto inmutables
    private final LocalDate fecha; // Fecha para usar el ticket
    private final String id; // Id del ticket

    // Atributos de objeto variables (estado)
    private LocalTime usado; // Momento de uso del ticket
}
```

Respecto a los posibles **constructores**, el equipo de desarrollo ha pensado en las siguientes posibilidades:

- 1.- un **constructor sin parámetros**, donde se genere un nuevo ticket para ser usado en la **fecha actual** (hoy);
- 2.- un **constructor con un parámetro** que indique la fecha para la que se podrá usar el ticket. Esa fecha no podrá ser anterior a la actual (pues ya no podría usarse) y debe estar dentro del año actual. No se pueden generar tickets para ser

usados en años futuros.

Implementa esos constructores en Java.

[Mostrar retroalimentación](#)

En la implementación de los constructores hay que poner en marcha la maquinaria que:

- 1.- compruebe que **aún se pueden generar tickets** (no se ha llegado al máximo número de la secuencia);
- 2.- compruebe **si se ha cambiado de año desde que se generó el último ticket**. Si no es así (el año registrado no es el año actual), hay **reiniciar la secuencia** para el nuevo año;
- 3.- **incremente el número de secuencia para generar el nuevo ticket** y haga que quede registrado en la clase (atributo estático) para la próxima vez que se vuelva a generar otro ticket.

Además, en la implementación del **constructor con un parámetro** habría que **comprobar que la fecha es válida (no null)**, que **no es anterior a la fecha actual** y que **está dentro del año actual**.

Podríamos realizar todas esas acciones y comprobaciones en el constructor de un parámetro:

```
public Ticket(LocalDate fecha) throws IllegalArgumentException, IllegalStateException {
    LocalDate hoy = LocalDate.now();
    if (fecha == null) {
        throw new IllegalArgumentException("fecha inválida (null)");
    }
    if (fecha.isBefore(hoy)) {
        throw new IllegalArgumentException("fecha de uso anterior a la actual");
    }
    if (fecha.getYear() > hoy.getYear()) {
        throw new IllegalArgumentException("fecha de uso posterior al año actual");
    }
    if (Ticket.lastSequence == Ticket.MAX_SEQUENCE) {
        throw new IllegalStateException("expedidos el número máximo de tickets para este año");
    }

    // Inicializamos atributos de objeto
    this.usado = null; // El ticket aún no ha sido usado
    this.fecha = fecha; // Registraremos la fecha en que podrá ser usado

    // Comprobamos si se ha cambiado de año desde la generación del último ticket
    if (Ticket.lastYear != hoy.getYear()) {
        Ticket.lastSequence = 0; // Se reinicia la secuencia
        lastYear= hoy.getYear(); // Se registra el nuevo año
    }
    // Generamos el id del ticket e incrementamos el número de secuencia
    id = String.format("%04d-%08d", hoy.getYear(), ++Ticket.lastSequence);
}
```

Y de este modo la implementación del **constructor sin parámetros** quedaría reducida a la mínima expresión (una invocación al constructor de un parámetro con la fecha actual usando `this()`):

```
public Ticket() throws IllegalStateException {
    this(LocalDate.now());
}
```

En cuanto a posibles métodos getter se ha pensado en los siguientes:

- ✓ uno que devuelva el **identificador** (id) del ticket (`getId`);
- ✓ otro que devuelva la **fecha** en la que se puede utilizar el ticket (`getFecha`);
- ✓ otro que indique **si ha sido ya utilizado** o no (`isUsado`);
- ✓ otro que indique si ticket es para ser utilizado en **fin de semana** (`isFinDeSemana`).

Implementa esos métodos de obtención de información.

[Mostrar retroalimentación](#)

La implementación de los getter podría quedar más o menos así:

```
public LocalDate getFecha() {
    return this.fecha;
}

public String getId() {
    return this.id;
}
```

```

public boolean isUsado() {
    return this.usado != null;
}

public boolean isFinDeSemana() {
    return (fecha.getDayOfWeek() == DayOfWeek.SATURDAY || fecha.getDayOfWeek() == DayOfWeek.SUNDAY);
}

```

Cuando un ticket vaya a ser **utilizado** podrá ejecutarse el método `usar`, que tendrá que comprobar:

- 1.- si nos encontramos **en la fecha de uso del ticket** (la fecha actual coincide con la fecha de uso);
- 2.- si el ticket **ha sido ya usado o no**.

En caso de que el ticket sea "usable", entonces habrá que **marcarlo como usado y registrar la hora** (hora y minuto) en la que fue usado.

[Mostrar retroalimentación](#)

Como ya comentamos a la hora de analizar los posibles atributos de la clase, podrían usarse dos atributos para el control del uso: un booleano que "marca" el uso del ticket y un LocalTime que registra la hora de uso. Ahora bien, también dijimos que podríamos utilizar simplemente la hora como marca (si aún tiene el valor null es que aún no se ha usado). Incluso también comentamos que en lugar de almacenar un Localtime podríamos directamente almacenar un String representando la hora con el formato "*hh:mm*". Cualquier alternativa sería razonable.

Nosotros hemos escogido la opción de usar un único atributo de tipo LocalTime.

También debes recordar que **un ticket, una vez usado, no puede volver a usarse**. Una forma de controlarlo podría ser lanzando una excepción de tipo `IllegalStateException` si se intenta usar un ticket ya marcado.

```

public void usar() throws IllegalStateException {
    if (!this.fecha.isEqual(LocalDate.now())) {
        throw new IllegalStateException (
            String.format("el ticket no es para hoy: %s",
            String.format ("%02d%02d%02d",
            this.fecha.getDayOfMonth(), this.fecha.getMonthValue(), this.fecha.getYear())));
    }

    if (isUsado()) {
        throw new IllegalStateException(
            String.format("el ticket ya ha sido usado: %s", this.usado));
    }

    // Si es posible usar el ticket lo marcamos como usado
    usado= LocalTime.now();
}

```

Respecto a una posible **representación textual** de un objeto Ticket se ha propuesto que se muestre la siguiente información:

- ✓ **identificador** (id) del ticket;
- ✓ **fecha** en que el ticket podrá ser usado;
- ✓ si no ha sido **usado** (cadena "no") o bien la hora a la que se usó (en formato "*hh:mm*").

El **formato** sería el siguiente:

{ID: NNNN-XXXXXXX, Fecha:dd/mm/aaaa, Usado: no | hora}

Algunos ejemplos de salidas de podrían ser:

```

{ID: 2020-00000003, Fecha:15/01/2020, Usado: 17:10}
{ID: 2020-00000301, Fecha:21/02/2020, Usado: 09:35}
{ID: 2020-00111111, Fecha:15/07/2020, Usado: no}
{ID: 2020-00921121, Fecha:26/04/2020, Usado: 21:57}
{ID: 2020-00001212, Fecha:12/03/2020, Usado: no}
{ID: 2020-99999999, Fecha:12/09/2020, Usado: no}

```

Implementa el método `toString` para que produzca una salida de ese estilo.

[Mostrar retroalimentación](#)

La implementación del método `toString` podría quedar así:

```

@Override
public String toString () {
    return String.format ("{ID: %s, Fecha:%02d/%02d/%02d, Usado: %s}",
        this.id,
        this.fecha.getDayOfMonth(), this.fecha.getMonthValue(), this.fecha.getYear(),
        this.usado == null ? "no" : String.format ("%02d:%02d", usado.getHour(), usado.getMinute()));
}

```

Por último, también se ha requerido la implementación de un par de métodos "fábrica" (o pseudoconstructores) que generen un ticket "aleatorio":

- ✓ un método randomEsteMes, que genere nuevo ticket para ser usado en una **fecha aleatoria del mes actual**;
- ✓ un método random, que genere un nuevo ticket para ser usado en una **fecha aleatoria del año actual**.

Implementa ambos métodos generadores de nuevos objetos Ticket.

[Mostrar retroalimentación](#)

Lo razonable es que estos métodos sean **estáticos** o **de clase** (static) pues no van a usar el contenido de ningún atributo de objeto. Son métodos "herramienta" que se utilizarán para generar nuevos tickets de manera similar a como hacemos con los constructores. De hecho en el interior estos métodos se va a producir:

1. la **generación de una fecha aleatoria** acorde con las características buscadas (que no sea anterior a la fecha actual y que esté dentro del mes o del año actual);
2. la **creación de un nuevo objeto** Ticket con esa **fecha aleatoria** mediante una **invocación al constructor** a través del operador new;
3. la **devolución de la referencia al objeto** instanciado devuelta por el operador new con la llamada al constructor.

La forma de generar una fecha aleatoria apropiada tendrás que llevarla a cabo jugando con las distintas posibilidades que te ofrecen los métodos de la clase LocalDate. Puedes enfocarlo desde distintas perspectivas.

Aquí tienes una posible implementación:

```

public static Ticket randomEsteMes () {
    LocalDate hoy= LocalDate.now(); // Fecha actual

    // Calculo día de mes actual
    int diaActual= hoy.getDayOfMonth();

    // Calculo último día de mes del mes actual
    int ultimoDiaMes= hoy.lengthOfMonth();

    // Calculo número aleatorio entre ambos
    int diasRandom= (int) (Math.random() * (ultimoDiaMes - diaActual +1)) ;

    // Generamos fecha del mes actual con día aleatorio y a partir de hoy
    LocalDate fechaRandom= hoy.plusDays(diasRandom);

    // Generamos ticket con la fecha aleatoria (dentro del mes actual)
    Ticket ticket= new Ticket (fechaRandom);

    // Devolvemos referencia a objeto ticket con fecha aleatoria
    return ticket;
}

public static Ticket random() {
    LocalDate hoy= LocalDate.now(); // Fecha actual

    // Calculo día del año para hoy (entre 1-365 o 366)
    int diaActual= hoy.getDayOfYear();

    // Calculo último día del año actual (365 o 366)
    int ultimoDia= 365 + (hoy.isLeapYear() ? 1 : 0);

    // Calculo número aleatorio entre ambos
    int diasRandom= (int) (Math.random() * (ultimoDia - diaActual +1)) ;

    // Generamos fecha del año actual con día aleatorio y a partir de hoy
    LocalDate fechaRandom= hoy.plusDays(diasRandom);

    // Generamos ticket con la fecha aleatoria (dentro del año actual)
    Ticket ticket= new Ticket (fechaRandom);

    // Devolvemos objeto ticket con fecha aleatoria
    return ticket;
}

```


I.3.- Clase Dado

Ejercicio Resuelto

Hemos recibido el encargo de implementar una clase que represente a los **dados de un juego de mesa**. Los posibles dados para estos juegos son todos aquellos poliedros que tengan todas las caras iguales. Matemáticamente existen cinco tipos diferentes (con cuatro, seis, ocho, doce o veinte caras). Son conocidos como los sólidos platónicos.

Para **cada dado** queremos registrar la siguiente información:

- ✓ el **número de caras** que tiene;
- ✓ la **cantidad de lanzamientos** que se han realizado con él.



Peng (Dominio público)

Teniendo en cuenta esas especificaciones, realiza un análisis inicial para decidir qué atributos necesitarías para implementar la clase Dado.

[Mostrar retroalimentación](#)

Se trata de una clase bastante simple para la cual, en principio, nos bastaría con un par de **atributos de objeto privados**:

- ✓ el **número de caras**, de tipo **entero** (por ejemplo byte), **inmutable** (el número de caras no va a cambiar);
- ✓ el **número de lanzamientos**, de tipo **entero** (por ejemplo long) y **mutable** (variable).

La implementación en Java podría quedar así:

```
public class Dado {
    // Atributos de objeto inmutables
    private final byte numeroCaras;

    // Atributos de objeto mutables (estado)
    private long numeroLanzamientos;
}
```

Para poder crear instancias de esta clase se nos ha sugerido implementar **dos constructores**: uno **con un parámetro entero**, que indique el **número de caras** del dado, y otro **sin parámetros**, que instancie un dado de seis caras.

Implementa ambos constructores.

[Mostrar retroalimentación](#)

En la implementación del primer constructor es necesario **asegurarse de que el número de caras pasado como parámetro es válido**. Si es así, entonces puede inicializarse el atributo que contiene el número de caras. Si no es así, se puede lanzar una **excepción** de tipo `IllegalArgumentException`:

```
public Dado (byte numeroCaras) throws IllegalArgumentException {
    // Comprobamos la validez del número de caras
    if (numeroCaras != 4 && numeroCaras != 6 && numeroCaras != 8 && numeroCaras != 12 && numeroCaras != 20) {
        throw new IllegalArgumentException (String.format ("número de caras no válido: %d", numeroCaras));
    }

    // Inicializamos los atributos de objeto
    this.numeroCaras= numeroCaras;
    this.numeroLanzamientos= 0;
}
```

El **constructor sin parámetros** puede implementarse como una simple llamada al constructor con parámetro con la **cantidad de caras por omisión** (seis). Si queremos evitar el uso de literales en los métodos, podemos declarar un **atributo constante de clase** que contenga ese valor:

```
public static final byte DEFAULT_NUMERO_CARAS= 6;
```

De ese modo el constructor **sin parámetros** quedaría así de simple:

```
public Dado () {
    this (Dado.DEFAULT_NUMERO_CARAS);
}
```

Respecto a los métodos get se ha decidido disponer de métodos para obtener la siguiente información:

- ✓ el **número de caras**, del dado;
- ✓ el **número de lanzamientos** que se han realizado con el dado.

Implementa ambos métodos.

[Mostrar retroalimentación](#)

Se trata de dos métodos get "directos" que no requieren ningún tipo de filtro ni transformación. Se trata simplemente de devolver el valor de un par de atributos:

```
public byte getNumeroCaras () {
    return this.numeroCaras;
}

public long getNumeroLanzamientos() {
    return this.numeroLanzamientos;
}
```

Para que los objetos de la clase Dado resulten de utilidad, han de poder lanzarse para obtener un resultado al azar. Se ha planteado un método llamado lanzar que devuelva un valor aleatorio equiprobable dentro de las caras que tenga el dado. Se nos ha pedido además el valor devuelto sea de tipo String y consista en una cadena que represente el número de la cara en castellano. Es decir, que este método devolverá cadenas del tipo "UNO", "DOS", "TRES", "CUATRO", etc.

Implementa en Java el método lanzar tal y como se ha especificado.

[Mostrar retroalimentación](#)

Para facilitar el tratamiento de los nombres de los números para cada cara lo más sencillo y operativo es almacenar esos nombres dentro de un array para poder obtenerlos con facilidad.

Ese array basta con que exista una sola vez y pueda ser usado por todos los objetos Dado que se vayan instanciando. Se trata por tanto de un array candidato a ser declarado como un atributo de clase. Será además constante o inmutable, pues no tiene mucho sentido que pueda cambiar. El array contendrá todos los nombres posibles y dependiendo del número de caras del dado se utilizarán todos o solo parte de esos nombres:

```
private static final String[] NOMBRES_CARAS= {"UNO", "DOS", "TRES", "CUATRO", "CINCO", "SEIS", "SIETE", "OCHO", "NUEVE",
"DIEZ", "ONCE", "DOCE", "TRECE", "CATORCE", "QUINCE", "DIECISEIS", "DICECISIETE", "DIECIOCHO", "DIECINUEVE", "VEINTE"};
```

En este caso lo podemos declarar como **privado** pues se trata de información "interna" que no va a resultar de utilidad a otros programadores que utilicen objetos de esta clase desde fuera.

Una vez que dispongamos del array con los nombres de los números, el método lanzar quedaría bastante sencillo:

```
public String lanzar() {
    byte aleatorio= (byte) (Math.random()*this.numeroCaras);
    return Dado.NOMBRES_CARAS[aleatorio];
}
```

En cuanto a una posible **representación textual** de los objetos de la clase Dado, se nos ha propuesto el siguiente formato de salida:

Número de caras: xxx. Número de lanzamientos: zzz

donde xxx será el número de caras del dado y zzz el número de veces que ha sido lanzado hasta el momento. Algunos ejemplos de salida podrían ser:

```
Número de caras: 6. Número de lanzamientos: 0
Número de caras: 12. Número de lanzamientos: 3
Número de caras: 4. Número de lanzamientos: 21
```

Teniendo en cuenta esas especificaciones, implementa un método `toString` en Java para la clase Dado.

[Mostrar retroalimentación](#)

La implementación del método `toString` podría quedar así:

```
@Override
public String toString() {
    return String.format ("Número de caras: %d, Número de lanzamientos: %d", this.numeroCaras, this.numeroLanzamientos);
}
```

Ejercicio Resuelto

Dado, nos ha llegado un nuevo requerimiento para que cada dado lleve un **registro del número de veces que ha salido cada cara**.

¿Cómo afectaría esta nueva funcionalidad a la nueva clase Dado desde el punto de vista de los atributos?

[Mostrar retroalimentación](#)

Una posible forma de registrar cuántas veces ha salido cada cara del dado podría ser mediante el uso de un **array de enteros de tamaño el número de caras del dado**. En cada casilla podría almacenar la cantidad de veces que ha salido cada cara. Sería un atributo **de objeto privado** de tipo **array de enteros** (por ejemplo int):

```
private int[] registroLanzamientos;
```

A partir de ese array podríamos obtener tanto el **número de caras** como la **cantidad de lanzamientos** pudiéndose eliminar esos dos atributos:

- ✓ el **número de caras** podría obtenerse a partir del **tamaño del array**: `registroLanzamientos.length`,
- ✓ la **cantidad de lanzamientos** podría calcularse **sumando todos los elementos del array**.

La declaración de atributos de esta nueva clase Dado podría quedar entonces así:

```
public class Dado {

    // ATRIBUTOS DE CLASE

    // Atributos constantes de clase
    public static final byte DEFAULT_NUMERO_CARAS= 6;

    private static final String[] NOMBRES_CARAS= {"UNO", "DOS", "TRES", "CUATRO", "CINCO",
    "SEIS", "SIETE", "OCHO", "NUEVE", "DIEZ", "ONCE", "DOCE", "TRECE", "CATORCE", "QUINCE",
    "DIECISEIS", "DICECISIETE", "DIECIOCHO", "DIECINUEVE", "VEINTE"};

    // ATRIBUTOS DE OBJETO
    private int[] registroLanzamientos;

    // (ya no necesitamos los atributos numeroCaras y numeroLanzamientos)
}
```

Teniendo en cuenta que ahora debemos registrar cada lanzamiento, ¿cómo implementarías el método lanzar?

[Mostrar retroalimentación](#)

En la implementación del método `lanzar` habrá que incluir la actualización de la posición apropiada del array `registroLanzamientos`: `this.registroLanzamientos[aleatorio]++`

```
public String lanzar() {
    byte aleatorio= (byte) (Math.random()*this.numeroCaras); // Calculamos cara aleatoria
    this.registroLanzamientos[aleatorio]++; // Incrementamos el registro de la cara obtenida
    return Dado.NOMBRES_CARAS[aleatorio]; // Devolvemos el nombre de la cara obtenida
}
```

Dado que ya no disponemos de los atributos `numeroCaras` y `numeroLanzamientos`, ¿cómo implementarías ahora los métodos `getNumeroCaras` y `getNumeroLanzamientos`?

[Mostrar retroalimentación](#)

Para el **número de caras** basta con obtener el tamaño del array registroLanzamientos:

```
public byte getNumeroCaras () {
    return (byte) this.registroLanzamientos.length;
}
```

Para calcular el **número total de lanzamientos** habrá que sumar el registro de veces que ha salido cada cara, es decir, sumar el contenido de cada uno de los elementos del array registroLanzamientos:

```
public long getNumeroLanzamientos() {
    long totalLanzamientos=0;
    for (int i=0; i<this.registroLanzamientos.length; i++) {
        totalLanzamientos += this.registroLanzamientos[i];
    }
    return totalLanzamientos;
}
```

Añade los cambios oportunos al **constructor** para la nueva clase Dado<code>.

[Mostrar retroalimentación](#)

Habrá que incluir la **reserva de memoria para el array de registro de lanzamientos** y desaparece la inicialización de los atributos numeroCaras<code> y numeroLanzamientos<code>, que ya no existirán.

```
public Dado (byte numeroCaras) throws IllegalArgumentException {
    // Comprobamos la validez del número de caras
    if (numeroCaras != 4 && numeroCaras != 6 && numeroCaras != 8 && numeroCaras != 12 && numeroCaras != 20) {
        throw new IllegalArgumentException (
            String.format ("numero de caras no válido: %d", numeroCaras));
    }

    // Inicializamos los atributos de objeto (reservamos espacio para el array de registros de cada cara)
    this.registroLanzamientos= new int[numeroCaras];
}
```

Dado que ahora que cada dado contiene un registro del número de veces que ha salido cada cara, estaría bien incorporar un nuevo método getNumeroVecesCara<code> que:

- 1.- reciba como **parámetro** un número entero que indique una **cara del dado** (entre 1 y el número de caras);
- 2.- devuelva el **número de veces que ha salido esa cara**.

Si el **parámetro** es un **valor inválido** (menor que 1 o superior al número de caras) debería lanzarse una excepción de tipo `IllegalArgumentException<code>`.

Implementa el método getNumeroVecesCara<code>.

[Mostrar retroalimentación](#)

La implementación del método getNumeroVecesCara<code> podría quedar así:

```
public int getNumeroVecesCara (byte cara) throws IllegalArgumentException {
    if (cara < 1 || cara > this.getNumeroCaras()) {
        throw new IllegalArgumentException ("numero de cara no válida");
    }
    return this.registroLanzamientos[cara-1];
}
```

I.4.- Clase Bombo

Ejercicio Resuelto

Nuestra empresa ha entrado en contacto con un posible cliente dedicado a los juegos de azar tradicionales que quiere lanzar una nueva línea de productos digitales. Uno de los primeros productos será un **bingo digital**.

El equipo de desarrollo ha recibido el encargo de representar un **bombo de un bingo**. Para ello habrá que implementar una clase Bombo cuyos objetos contendrán una serie de bolas que podrán ir extrayéndose una a una con la particularidad de que una vez que se extraiga una bola ya no podrá volver a salir.

El **número de bolas** con el que podemos configurar los bombos estará entre **9 y 90**.



Nina Garmaa (Pixabay License)

Teniendo en cuenta esas especificaciones, realiza un análisis preliminar sobre cuáles serían los atributos apropiados para implementar la clase Bombo.

[Mostrar retroalimentación](#)

Para poder trabajar con objetos de tipo Bombo tal y como se nos propone debemos tener en cuenta que:

- ✓ un **bombo lleno** puede tener **entre 9 y 90 bolas**, según se haya configurado (probablemente se indicará en el **constructor**);
- ✓ según se vayan **extrayendo bolas**, la **cantidad de elementos en el bombo se irá decrementando** hasta quedar vacío (cero bolas);
- ✓ cada bola que se extraiga (un número entero) **no puede repetirse**.

Eso significa que vamos a necesitar llevar un **registro** o bien de las **bolas (números) que ya han salido** o bien de las **bolas que quedan por salir**. Dado que la cantidad de números que necesitamos registrar va a estar entre 9 y 90, no es razonable declarar noventa atributos para poder ir "marcando" bolas. Para este tipo de situaciones tenemos estructuras de datos más complejas que nos permiten manejar grandes volúmenes de datos homogéneos: los **arrays**.

La forma en que podemos manejar esa información mediante un array permite diversos enfoques. Por ejemplo:

- ✓ disponer de un array de boolean de tamaño entre 9 y 90 (según el tamaño del bombo) donde se vayan marcando las bolas que van saliendo. De esa manera, cada vez que se genere un número aleatorio para simular una nueva extracción, podemos comprobar si ese número ya ha salido y si es así, se vuelve a generar un nuevo número.
- ✓ disponer de un array similar pero de números enteros y en lugar de marcar con true y false, marcar con 0 y 1.
- ✓ gestionar un array donde se vayan guardando los números que ya han salido o bien los que quedan por salir. Cada vez que se genere un nuevo número aleatorio para realizar una extracción, podemos recorrer ese array para comprobar que el candidato a salir es válido (aún no ha salido).
- ✓ etc.

Para gestionar los números extraídos hemos ideado el siguiente algoritmo:

1. rellenamos un array de enteros con los números desde 1 hasta el número de bolas;
2. guardamos en otro atributo entero el número de bolas que se llevan extraídas hasta el momento (cero al empezar);
3. cada vez que se quiera extraer una bola del bombo se genera un aleatorio entre 0 y el número de bolas que quedan por salir menos 1. Se toma el número que haya en esa posición del array y se intercambia con el número que haya en la posición "número de bolas que quedan por salir menos 1". Se incrementa el número de bolas que se llevan extraídas;
4. la próxima vez que se quiera extraer otra bola, el rango de aleatorios a generar será un poco más pequeño (un número menos) y ya no se podrá extraer el último elemento del array (pues ahí se colocó el número que salió en la extracción anterior). De esta manera se irá estrechando el margen de números aleatorios que pueden generar y aumentando la cantidad de números que ya han salido, ambos conjuntos almacenados en el mismo array y separados por ese contador de número de bolas extraídas hasta el momento.

Es un poco más complejo que las otras soluciones, pero incorpora importantes ventajas:

1. cada vez que generemos un número aleatorio, buscaremos en la zona del array con bolas por salir, de manera que siempre se obtendrá un número de bola que se puede extraer (no hay que hacer ninguna nueva comprobación ni volver a generar otro aleatorio);
2. tenemos a la vez, en un mismo array, las bolas que ya han salido y las que quedan por salir.

Si decidimos trabajar con ese esquema para la simulación del comportamiento del bombo de un bingo necesitaríamos tan solo **dos atributos de objeto**:

1. El **array de enteros con todos números de las bolas** del bombo. Aunque el contenido en cada casilla del array podrá variar, el array en sí (zona de memoria) no cambiará. Una vez que se reserve memoria para él en el constructor mediante el operador new, esa referencia ya no cambiará. Podríamos declararlo final si lo deseamos. Pero no olvides que lo que será **constante** es la propia **estructura del array** (la zona de memoria en la que se encuentra y su tamaño), pero su contenido seguirá siendo variable.
2. Un **entero** que indique **cuántas bolas llevamos extraídas** hasta el momento. Atributo **variable**.

Además de esos atributos de objeto, también podríamos definir algunos atributos estáticos constantes para definir rangos y valores por omisión:

- ✓ **mínimo y máximo número de bolas** para crear un bombo (9 y 90);

- ✓ tamaño (número de bolas) por omisión para un bombo (por ejemplo 90).

Una vez que tenemos todo esto claro, la declaración de los atributos de la clase Bombo podría quedar así:

```
public class Bombo {

    // ATRIBUTOS DE CLASE
    // -------

    // Atributos de clase constantes
    public static byte MINIMA_CAPACIDAD = 9;
    public static byte MAXIMA_CAPACIDAD = 90;
    public static final byte DEFAULT_CAPACIDAD = Bombo.MAXIMA_CAPACIDAD;

    // ATRIBUTOS DE OBJETO
    // -------

    // No es necesario un atributo para la capacidad del bombo (es el tamaño del array)
    private final int[] listaBolas; // Los elementos del array serán variables, el array siempre será el mismo (final)
    private int cantidadBolasExtraidas;
}
```

Se han previsto dos constructores para esta clase:

- 1.- un **constructor con un parámetro**, donde se indicará la **capacidad del bombo** (número de bolas cuando esté lleno);
- 2.- un **constructor sin parámetros**, que creará un bombo con la capacidad por omisión (noventa bolas).

[Mostrar retroalimentación](#)

En la implementación del **constructor con un parámetro** habrá que tener en cuenta que la **capacidad** pasada como parámetro esté dentro del **rango válido**. Si no se cumple, habrá que lanzar una excepción de tipo `IllegalArgumentException`.

Una vez comprobado que la capacidad es válida, habrá que reservar espacio para el array y rellenarlo con números enteros consecutivos desde el 1 hasta la capacidad del bombo. Ese relleno podríamos realizarlo en un método auxiliar privado por si lo necesitamos usar más adelante:

```
private void llenar () {
    for (int i = 0; i < this.getCapacidad(); i++) {
        listaBolas[i] = i + 1;
    }
}
```

Usado ese método auxiliar, el **constructor con un parámetro** podría quedar así:

```
public Bombo(int capacidad) throws IllegalArgumentException {
    if (capacidad < Bombo.MINIMA_CAPACIDAD || capacidad > Bombo.MAXIMA_CAPACIDAD) {
        throw new IllegalArgumentException("capacidad de bombo no válida: " + capacidad);
    } else {
        listaBolas = new int[capacidad];
        llenar();
    }
}
```

El **constructor sin parámetros** quedaría reducido a una simple llamada a `this()` con el valor de la capacidad por omisión:

```
public Bombo() {
    this(Bombo.DEFAULT_CAPACIDAD);
}
```

Como métodos get para esta clase se ha pensado en aquellos que permitan:

- ✓ obtener la **capacidad del bombo** (número de bolas cuando está lleno): `getCapacidad`;
- ✓ obtener la **cantidad de bolas que quedan por salir**: `getCantidadBolasRestantes`;
- ✓ obtener la **cantidad de bolas que ya han sido extraídas**: `getCantidadBolasExtraidas`;
- ✓ indicar **si el bombo está completamente lleno**: `isCompleto`;
- ✓ indicar **si el bombo está completamente vacío**: `isVacio`.

Implementa en Java esa lista de métodos.

[Mostrar retroalimentación](#)

Se trata de un conjunto de métodos bastante sencillos de implementar:

```
public int getCapacidad() {
    return this.listaBolas.length;
}

public int getCantidadBolasExtraidas() {
    return this.cantidadBolasExtraidas;
}

public int getCantidadBolasRestantes() {
    return this.getCapacidad() - this.cantidadBolasExtraidas;
}

public boolean isEmpty() {
    return this.cantidadBolasExtraidas == this.capacidad;
}

public boolean isComplete() {
    return this.cantidadBolasExtraidas == 0;
}
```

En cuanto a una posible **representación textual** de los objetos Bombo, se ha planteado el siguiente formato:

Capacidad: xx bolas. Cantidad de bolas extraídas: zz

donde xx será la capacidad del bombo (número de bolas cuando está lleno) y zz el número de bolas que han sido extraídas hasta el momento. Algunos ejemplos de salida podrían ser:

Capacidad: 90 bolas. Cantidad de bolas extraídas: 0
 Capacidad: 9 bolas. Cantidad de bolas extraídas: 9
 Capacidad: 10 bolas. Cantidad de bolas extraídas: 7
 Capacidad: 90 bolas. Cantidad de bolas extraídas: 63

Teniendo en cuenta esas especificaciones, implementa un método `toString` Java para la clase Bombo.

[Mostrar retroalimentación](#)

La implementación del método `toString` para generar esa salida podría quedar así:

```
@Override
public String toString() {
    return String.format("Capacidad: %d bolas. Cantidad de bolas extraídas: %d",
        this.getCapacidad(), this.cantidadBolasExtraidas);
}
```

El método "estrella" de esta clase es el de **extracción de una bola**, que debe evitar que se repitan bolas. **Si el número de bolas que quedan por salir es cero**, debería lanzarse una **excepción**, pues no se puede devolver ningún número.

[Mostrar retroalimentación](#)

Como ya comentamos durante el análisis inicial de atributos, existen diversos enfoques para gestionar la forma de evitar la repetición de números al extraer una bola. Nosotros en su momento ya nos decidimos por una. Teniendo en cuenta ese algoritmo, el método de extracción de bola quedaría así:

```
public int extraerBola() throws IllegalStateException {
    if (this.isEmpty()) {
        throw new IllegalStateException("bombo vacío");
    } else {
        int aleatoria = this.cantidadBolasExtraidas
            + (int) (Math.random() * (this.getCapacidad() - this.cantidadBolasExtraidas));
        int bola = listaBolas[aleatoria]; // bola extraída
        listaBolas[aleatoria] = listaBolas[this.cantidadBolasExtraidas];
        listaBolas[this.cantidadBolasExtraidas] = bola;
        this.cantidadBolasExtraidas++;
        return bola;
    }
}
```

Este algoritmo tiene cierta complejidad. Si no comprendes su funcionamiento te recomendamos que hagas un pequeño dibujo con un array de 9 elementos relleno con los valores del 1 al 9 y vayas ejecutando una vez tras otra el método para observar su funcionamiento. Es una manera muy eficiente y elegante de gestionar la generación de números aleatorios para evitar su repetición.

También se nos ha pedido un método llamado reset, sin parámetros, que reinicie el bombo y lo vuelva a llenar con todas las bolas para empezar de nuevo. Este método además devolverá un entero que será la cantidad de bolas que ha habido que volver a introducir en el bombo.

Implementa el método reset.

[Mostrar retroalimentación](#)

Si hemos implementado anteriormente el método auxiliar llenar, este método puede aprovecharlo invocándolo para poner el array de bolas en orden. Respecto al valor que debe devolver (cantidad de bolas con las que hay que llenar) es simplemente la cantidad de bolas que han sido extraídas hasta el momento:

```
public int reset () {
    int bolasFuera= this.cantidadBolasExtraidas;
    this.cantidadBolasExtraidas = 0;
    // Rellenamos la lista de Bolas ordenadamente
    llenar ();
    return bolasFuera;
}
```

Ejercicio Resuelto

Una vez que disponemos de la clase Bombo, estaría bien llevar a cabo algunas pruebas mediante algún programa (clase de pruebas con método main) que se asegure de que todo funciona correctamente.



[Nina Garman \(Pixabay License\)](#)

Implementa un programa de prueba para la clase Bombo que lleve a cabo las siguientes acciones:

- 1.- Cree un bombo con la capacidad por omisión y muestre su estado (método `toString`).
- 2.- Solicite por teclado un número entero como capacidad para el bombo.
- 3.- Intente crear un bombo con esa capacidad. Si no es posible porque la capacidad no es válida (salta una excepción) o bien porque se introduce algún valor inapropiado como número entero, entonces el programa finaliza.
- 4.- Si el bombo ha podido ser creado correctamente, mostramos el estado del bombo (método `toString`).
- 5.- A continuación se ejecutará un bucle en el que se irán realizando extracciones de bolas y se irán mostrando por pantalla hasta que salte la excepción por bombo vacío.
- 6.- Una vez que el bombo esté vacío se muestra el estado del bombo (método `toString`).

Una posible salida por pantalla del resultado de la ejecución del programa podría ser algo así:

```
BOMBO DE UN BINGO
-----
Prueba del constructor sin parámetros:
Creado objeto: Capacidad: 90 bolas. Cantidad de bolas extraídas: 0

Prueba del constructor con parámetros:
Introduzca el número de bolas (9-90): 10
Prueba del getNumBolas: 10
Prueba del getNumBolasExtraídas: 0
Prueba del getNumBolasRestantes: 10
Prueba del toString: Capacidad: 10 bolas. Cantidad de bolas extraídas: 0
Pruebas de extracción: 8 10 9 6 2 3 4 1 5 7
Error: bombo vacío.
Estado actual del bombo: Capacidad: 10 bolas. Cantidad de bolas extraídas: 10

Introduzca el número de bolas (9-90): 8
capacidad inválida: 8. Finalizamos
```

[Mostrar retroalimentación](#)

Un posible método main para una clase de pruebas podría ser el siguiente:

```
public static void main() {
    int numBolas;
    Bombo bombo = null;
    boolean entradaValida = true;
    Scanner teclado = new Scanner(System.in);

    System.out.println("BOMBO DE UN BINGO");
    System.out.println("-----");
    System.out.println("Prueba del constructor sin parámetros: ");
    bombo = new Bombo();
    System.out.println("Creado objeto: " + bombo);

    System.out.println("\nPrueba del constructor con parámetros: ");
    do {
        entradaValida = true;
        System.out.print("Introduzca el número de bolas (9-90): ");
        try {
            numBolas = teclado.nextInt();
            bombo = new Bombo(numBolas);
        } catch (IllegalArgumentException ex) {
            System.out.println(ex.getMessage() + ". Finalizamos");
            entradaValida = false;
        } catch (InputMismatchException ex) {
            System.out.println("Valor inválido. Finalizamos");
            entradaValida = false;
        }
    } while (entradaValida);
    System.out.println("Prueba del getNumBolas: " + bombo.getCapacidad());
    System.out.println("Prueba del getNumBolasExtraidas: " + bombo.getCantidadBolasExtraidas());
    System.out.println("Prueba del getNumBolasRestantes: " + bombo.getCantidadBolasRestantes());
    System.out.println("Prueba del toString: " + bombo.toString());
    System.out.print("Pruebas de extracción: ");

    try {
        for (; ; ) { // Bucle infinito de extracción de bolas hasta que "falle"
            System.out.printf("%d ", bombo.extraerBola());
        }
    } catch (IllegalStateException e) {
        System.out.printf("\nError: %s.", e.getMessage());
    }

    System.out.println("\nEstado actual del bombo: " + bombo);
    System.out.println();
}
}
```

Ejercicio Resuelto

Una vez que disponemos de una implementación básica y funcional de la clase Bombo, nos han pedido una ampliación para que incorpore dos nuevos métodos que permitan:

- 1.- obtener un array con los números de bola que quedan en el bombo;
- 2.- obtener un array con los números de bola que ya han salido del bombo.



Alejandro Garay (Pixabay License)

¿Cómo te plantearias añadir esta nueva funcionalidad? ¿Necesitarías atributos adicionales? ¿Tendrías que modificar la implementación de alguno de los métodos que ya tienes?

[Mostrar retroalimentación](#)

Si has implementado la clase con un método extraerBola que usa un array que contiene a la vez la lista de bolas pendientes de salir y también las que ya han salido, separadas por la cantidad de bolas que quedan por salir, implementar esos dos métodos es muy sencillo, pues se trata simplemente de obtener un subarray de ese array general.

Implementa el método public short[] getBolasExtraídas()

que devuelve un **array con los números de las bolas que ya han salido del bombo**.

[Mostrar retroalimentación](#)

Para obtener el array de bolas que ya han sido extraídas puede hacerse lo siguiente:

- ✓ construir un array que contenga tantos elementos como bolas hayan sido extraídas hasta el momento;
- ✓ copiar a ese array esa cantidad de elementos desde el array general comenzando por la posición inicial (0).

Eso significa que si aún no se ha extraído ninguna bola, el array generado será de tamaño 0 y si se han extraído todas las bolas, el array generado será de tamaño toda la capacidad del bombo (el array general completo).

Para extraer parte de un array podemos usar el método estático copyOfRange de la clase Arrays:

```
public int[] getBolasExtraídas() {
    int[] resultado = Arrays.copyOfRange(listaBolas, 0, this.cantidadBolasExtraídas);
    return resultado;
}
```

Implementa el método public short[] getBolasRestantes() que devuelve un **array con los números de las bolas que aún permanecen en el bombo**.

[Mostrar retroalimentación](#)

Para implementar este método se puede hacer algo similar a lo que se ha hecho en el método anterior.

Para extraer justo la parte contraria del array (desde la posición que indica la cantidad de bolas hasta el final). De este modo, si la cantidad de bolas extraídas es cero, el array obtenido será el total de todas las bolas y si la cantidad de bolas extraídas es el total, el array obtenido será de tamaño cero:

```
public int[] getBolasRestantes() {
    int[] resultado = Arrays.copyOfRange(listaBolas, this.cantidadBolasExtraídas, this.getCapacidad());
    Arrays.sort(resultado);
    return resultado;
}
```

Si además queremos que las bolas que queden por salir aparezcan ordenadas podemos usar el método estático sort de la clase Arrays.

Por último, nos quedaría implementar algún programa de prueba para comprobar que esta clase funciona correctamente. Podríamos escribir por ejemplo un programa que generara un bombo de capacidad nueve bolas, que fuera extrayendo las bolas una a una y que se fueran mostrando por pantalla la lista (array) de bolas extraídas y la de bolas restantes. La salida por pantalla podría ser algo así:

PRUEBAS DE BOMBO MEJORADO

Instanciando bombo de capacidad 9 bolas.

Bombo creado.

Estado inicial del bombo: Capacidad: 9 bolas. Cantidad de bolas extraídas: 0.

Extraídas: []

Restantes: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Pruebas de extracción:

Bola extraída: 9

Array de bolas extraídas: [9]

Array de bolas restantes: [1, 2, 3, 4, 5, 6, 7, 8]

Bola extraída: 1

Array de bolas extraídas: [9, 1]

Array de bolas restantes: [2, 3, 4, 5, 6, 7, 8]

Bola extraída: 8

Array de bolas extraídas: [9, 1, 8]

Array de bolas restantes: [2, 3, 4, 5, 6, 7]

Bola extraída: 3

Array de bolas extraídas: [9, 1, 8, 3]

Array de bolas restantes: [2, 4, 5, 6, 7]

Bola extraída: 6

Array de bolas extraídas: [9, 1, 8, 3, 6]

Array de bolas restantes: [2, 4, 5, 7]

Bola extraída: 2

Array de bolas extraídas: [9, 1, 8, 3, 6, 2]

Array de bolas restantes: [4, 5, 7]

Bola extraída: 5

Array de bolas extraídas: [9, 1, 8, 3, 6, 2, 5]

```
Array de bolas restantes: [4, 7]
Bola extraída: 7
Array de bolas extraídas: [9, 1, 8, 3, 6, 2, 5, 7]
Array de bolas restantes: [4]
Bola extraída: 4
Array de bolas extraídas: [9, 1, 8, 3, 6, 2, 5, 7, 4]
Array de bolas restantes: []
Error: Bombo vacío.
Estado final del bombo: Capacidad: 9 bolas. Cantidad de bolas extraídas: 9
```

Escribe un método main en Java para realizar una prueba similar a ésta.

[Mostrar retroalimentación](#)

```
public static void main() {
    Bombo bombo = null;

    System.out.println("BOMBO MEJORADO");
    System.out.println("-----");
    try {
        System.out.println("Instanciando bombo de capacidad 9 bolas.");
        bombo = new BomboAmpliado(9);
        System.out.println("Bombo creado.");
        System.out.printf("Estado inicial del bombo: %s.\n", bombo);
        System.out.printf("Extraídas: %s\n", Arrays.toString(bombo.getBolasExtraídas()));
        System.out.printf("Restantes: %s\n", Arrays.toString(bombo.getBolasRestantes()));
        System.out.printf("Pruebas de extracción: \n");
        try {
            for ( ; ; ) { // Bucle infinito hasta que no se puedan extraer más bolas
                System.out.printf("Bola extraída: %d\n", bombo.extraerBola());
                System.out.printf("Array de bolas extraídas: %s\n", Arrays.toString(bombo.getBolasExtraídas()));
                System.out.printf("Array de bolas restantes: %s\n", Arrays.toString(bombo.getBolasRestantes()));
            }
        } catch (IllegalStateException e) {
            System.out.printf("Error: %s.\n", e.getMessage());
        }
        System.out.println("Estado final del bombo: " + bombo);
    } catch (IllegalArgumentException ex) {
        System.out.println(ex.getMessage() + ".");
    }

    System.out.println();
}
```

Anexo II.- Implementación de la clase Vehiculo

A continuación te ofrecemos una tarea original propuesta hace algunos años para ver qué tal te desenvuelves a la hora de resolverla. Seguro que hay muchos elementos que te suenan a todo lo que hemos estado haciendo durante la unidad. ¡Suerte!

NOTA: debes tener en cuenta que no todos los métodos o los atributos que aquí se propongan para la implementación de una clase Vehiculo tienen por qué coincidir exactamente con los ejemplos que se han ido planteando a lo largo del contenido la unidad. En este ejercicio tendrás que atenerte a lo que se indique en las especificaciones concretas del enunciado de esta tarea.

Caso práctico

Una empresa de logística que es un potencial cliente de **BK Programación** ha manifestado su inquietud respecto a la posibilidad de automatizar la gestión de sus vehículos de transporte terrestre. Para ello será necesario desarrollar un prototipo de simulación del comportamiento de sus vehículos.

Ada ha repartido el trabajo entre los distintos componentes de la empresa, y a **Juan** le ha tocado evaluar la posibilidad de crear un simulador para observar cómo funcionan los vehículos. Se trataría de una aplicación muy sencilla que permitiera crear vehículos para observar su comportamiento respecto al consumo, recorridos, etc.

Juan está dándole vueltas a la idea pensando en los atributos y métodos que serán necesarios para modelar un vehículo de la manera más simple posible:



Shai Farley (CC BY-SA)

— Habría que disponer de atributos para almacenar el **tamaño del depósito** y el **consumo medio**, y éstos atributos ya serán inmutables a lo largo de la vida de todo el objeto. Sin embargo, otros atributos como la **distancia recorrida** o el **consumo producido**, estarán más relacionados con el estado del objeto y podrán ir cambiando a lo largo de la vida de éste, ¿no crees?

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

— ¡Claro! Imagina que queremos almacenar en alguna parte **todos los kilómetros recorridos por todos los coches**. ¿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 representa a los vehículos.

La tarea consiste en la implementación de un **pequeño programa para simular el funcionamiento elemental de una flota de vehículos**. La aplicación permitirá crear objetos de tipo Vehiculo con unas características determinadas:

- ✓ **capacidad del depósito de combustible** (en litros);
- ✓ **consumo medio del vehículo** (en litros/100 km).

Además, estos objetos de tipo Vehiculo contendrán **información de estado** como:

- ✓ si el vehículo tiene el **motor encendido o apagado**;
- ✓ el **nivel actual del depósito de combustible** (en litros);
- ✓ la **cantidad de kilómetros recorridos desde que se ha arrancado por última vez** (en kilómetros);
- ✓ el **consumo del vehículo desde que se ha arrancado por última vez** (en litros);
- ✓ la **cantidad total de kilómetros recorridos desde que se fabricó el vehículo** (en kilómetros);
- ✓ el **consumo total del vehículo desde que se fabricó** (en litros).



Mariordo (CC BY-SA)

Por otro lado, se pretende también que la propia clase, independientemente de la existencia o no de objetos, contenga **información global** del siguiente tipo:

- ✓ **distancia recorrida total por todos los vehículos que se hayan creado hasta el momento** (en kilómetros);
- ✓ **combustible total consumido por todos los vehículos que se hayan creado hasta el momento** (litros);
- ✓ **número de vehículos con el motor encendido en el momento actual**.

Las actividades de esta tarea se van a centrar en la implementación de una clase llamada Vehiculo, donde vamos a utilizar una gran parte de los mecanismos y recursos relacionados con el desarrollo de clases y objetos vistos en esta unidad (declaración de una clase, atributos variables, atributos constantes, atributos estáticos, atributos de objeto, constructores, getters y setters, métodos de objeto que cumplen funcionalidades específicas; métodos de clase o estáticos para un uso global e independientemente de los objetos, etc.).

II.1.- Descripción de la tarea

- Usando el IDE NetBeans, crea un proyecto con dos clases: la clase **Vehículo** y la clase de pruebas (llamada Principal) que se proporciona con esta tarea. En la clase **Vehículo** se definirán los atributos necesarios para representar las siguientes características de un vehículo:
 - ✓ estos dos primeros atributos tendrán un valor que se definirá al crearse el objeto y ya nunca cambiarán su valor durante toda la vida del objeto:
 - ◆ **capacidad del depósito** de combustible (en litros);
 - ◆ **consumo medio** del vehículo (en litros/100 km);
 - ✓ estos otros atributos sí irán cambiando a lo largo de la vida del objeto y de alguna manera representarán el estado del objeto en cada momento:
 - ◆ si el vehículo tiene el **motor encendido o apagado**;
 - ◆ el **nivel actual del depósito** de combustible (en litros);
 - ◆ la **cantidad de kilómetros recorridos** desde que se ha arrancado por última vez (en kilómetros);
 - ◆ el **consumo realizado** desde que se ha arrancado por última vez (en litros);
 - ◆ la **cantidad total de kilómetros recorridos** desde que se fabricó el vehículo (en kilómetros);
 - ◆ el **consumo total realizado** desde que se ha arrancado por última vez (en litros).
 - ✓ por último, también habrá otros atributos que pertenecerán a la clase, más que a un objeto en particular, y tendrán sentido siempre, independientemente de que existan o no instancias de la clase (atributos estáticos):
 - ◆ **distancia recorrida total por todos los vehículos** que se hayan ido creando hasta el momento (kilómetros);
 - ◆ **combustible total consumido por todos los vehículos** que se hayan ido creando hasta el momento (litros);
 - ◆ **número de vehículos con el motor encendido** en el momento actual.
 - ✓ además de esos atributos variables, la clase también va a contener **constants** que serán de utilidad a la hora de realizar comprobaciones o asignaciones por defecto:
 - ◆ **mínimo consumo medio** permitido a la hora de crear un nuevo vehículo (**2,0 litros/100 km**);
 - ◆ **máximo consumo medio** permitido a la hora de crear un nuevo vehículo (**20,0 litros/100 km**);
 - ◆ **mínima capacidad del depósito** permitida a la hora de crear nuevo vehículo (**10,0 litros**);
 - ◆ **máxima capacidad del depósito** permitida a la hora de crear nuevo vehículo (**120,0 litros**).
 - ◆ **consumo** que se produce **al arrancar un vehículo** (siempre será el mismo: **0,02 litros**).
 - ◆ **valor por omisión para el consumo medio** de un vehículo (**5,0 litros/100 km**).
 - ◆ **valor por omisión para la capacidad del depósito** de un vehículo (**50,0 litros**).
2. Implementa **dos métodos constructores** para la clase **Vehículo**:
- ✓ uno **sin parámetros** que creará una instancia de un vehículo con un **depósito de tamaño 50,0 litros** y un **consumo medio de 5,0 litros/100 km**. Esos valores deben aparecer como atributos constantes de la clase y es obligatorio usarlos en el constructor;
 - ✓ otro que permita crear un objeto de tipo **Vehículo** donde la capacidad del depósito y el consumo se le pasan como parámetros. Ahora bien, si los parámetros no son válidos (no cumplen las condiciones que se piden), entonces se lanzará una excepción de tipo **IllegalArgumentException** y el objeto no será creado. Las condiciones mínimas exigibles para que un objeto de tipo **Vehículo** pueda ser creado es que la capacidad del depósito se encuentre entre los 10 y los 120 litros (ambos inclusive) y el consumo medio esté entre 2,0 litros/100 km y 20 litros/100 km. Todas estas cifras estarán definidas en las constantes de clase que se han descrito en apartados anteriores y **deben utilizarse esas constantes para las comprobaciones en el constructor, y no valores literales**.
3. Añade a la clase **Vehículo** los **métodos consultores (getters)** para poder obtener la siguiente información:
- ✓ **isArrancado()**, que indique **si el motor del vehículo está arrancado o no**;
 - ✓ **getConsumoMedio()**, que devuelva el **consumo medio del vehículo** (en litros/100 km);
 - ✓ **getCapacidadDeposito()**, que devuelva la **capacidad del depósito de combustible del vehículo** (en litros);
 - ✓ **getNivelCombustible()**, que devuelva el **nivel actual del depósito de combustible del vehículo** (en litros);
 - ✓ **getDistanciaRecorrida()**, que devuelva la **distancia recorrida por el vehículo desde que ha sido arrancado por última vez** (en km);
 - ✓ **getDistanciaRecorridaTotal()**, que devuelva la **distancia total recorrida por el vehículo desde su fabricación** (en km);
 - ✓ **getCombustibleConsumido()**, que devuelva el **combustible que ha sido consumido por el vehículo desde que ha sido arrancado por última vez** (en litros);
 - ✓ **getCombustibleConsumidoTotal()**, que devuelva el **combustible total que ha sido consumido por el vehículo desde su fabricación** (en litros);
 - ✓ **getDistanciaRecorridaFlota()**, que devuelva la **distancia recorrida total por todos los vehículos que se hayan creado hasta el momento** (en km);
 - ✓ **getCombustibleConsumidoFlota()**, que devuelva el **combustible total que ha sido consumido por todos los vehículos que hayan sido creados hasta el momento** (en litros);
 - ✓ **getNumVehiculosArrancadosFlota()**, que devuelva el **número de vehículos que haya con el motor encendido** en ese momento.
4. Implementa un método repostar que permite **rellenar el depósito de combustible del vehículo**. Recibirá un parámetro que indicará la cantidad de litros para repostar. Este método **actualizará el nivel de combustible del vehículo en función del valor del parámetro que se le pase**. Ahora bien, podrían darse **algunos casos de error** que deberían lanzar una excepción:
1. **si el motor está arrancado**, no se podrá repostar, y se lanzará una excepción `<code>IllegalStateException`. El mensaje asociado a la excepción debería ser "se intentó repostar con el motor encendido. No se ha repostado.".
 2. **si la cantidad de combustible que se intenta repostar es superior a la que puede admitir el depósito en ese momento**, el depósito se llenará, pero al rebosar hará que se lance una excepción `IllegalArgumentException`. En este caso, el mensaje asociado a la excepción sería "depósito lleno. Se ha sobrepasado la capacidad del depósito de combustible en xxx litros.", donde xxx sería la cantidad en exceso de combustible que se ha intentado repostar y que no se ha podido (lo que habría rebosado).
5. Implementa un método arrancar que hace que **el estado del motor pase a encendido (o arrancado)**. Arrancar el motor de un vehículo hará que produzca una pequeña cantidad de **consumo** de combustible. El consumo que se produce al arrancar estará definido en una de las constantes de clase que se han descrito anteriormente y habrá que utilizar esa constante para modificar el estado del objeto (incrementar el consumo y decrementar el nivel del depósito). Ahora bien, pueden producirse algunas situaciones de error:
1. **el estado del motor pasa a arrancado si aún no lo está**, pero si ya lo está, no se producirá ningún consumo y además se lanzará una excepción de tipo `IllegalStateException` con el mensaje "Error: El motor ya se encuentra arrancado.";
 2. **si el depósito del vehículo no tiene combustible suficiente para poder arrancar, se producirá una excepción `IllegalStateException`** y se vaciará completamente lo que quede de combustible. El mensaje de la excepción será en este caso "depósito vacío. Se intentó arrancar sin combustible suficiente.".
6. Implementa un método **realizarTrayecto**, que haga que el vehículo realice un trayecto de una determinada cantidad de kilómetros que se pasarán como parámetro. Podrá generarse una situación de error, y por tanto el lanzamiento de una excepción, en alguno de los siguientes casos:



[OpenClipart-Vectors](#) (Licencia Pixabay)

1. excepción **IllegalArgumentException** si el valor que se pasa como distancia para recorrer es **negativo**. El mensaje de la excepción será en este caso "Error: Se intentó realizar un trayecto negativo.";
 2. excepción **IllegalStateException** si está el **motor apagado**. El mensaje de la excepción será en este caso "Error: Se intentó realizar un trayecto con el motor apagado. No se ha avanzado.";
 3. excepción **IllegalArgumentException** si no hay combustible suficiente para recorrer esa distancia. En tal caso se consume todo el depósito, se recorre la distancia que sea posible, y se apaga el motor. El mensaje de la excepción será en este caso "no se ha podido finalizar el trayecto completamente. Depósito vacío. Han faltado por recorrer xxx km.", donde xxx sería la cantidad de kilómetros que no se han podido recorrer debido a la falta de combustible.
 7. Implementa un método apagar, que **apague el motor** y reinicie a cero los indicadores de consumo y distancia recorrida desde el último arranque. Si el motor ya estuviera apagado, se debería lanzar una excepción de tipo **IllegalStateException** con un mensaje de error del tipo "el motor ya se encuentra apagado." y no llevar a cabo ninguna acción.
 8. Implementa un método ***toString*** que represente el estado de un vehículo generando un **String** que:
 1. contenga la siguiente **información del estado del vehículo** en ese momento:
 1. **estado del motor** (encendido o apagado);
 2. **nivel del depósito** de combustible;
 3. **distancia recorrida** desde que se ha arrancado (si es que el motor está arrancado, pues si está apagado será obviamente cero);
 4. **consumo realizado** desde que se ha arrancado (si es que el motor está arrancado, pues si está apagado será obviamente cero);
 2. La muestre con el siguiente formato: "**Motor: XXX - Deposito: YYY - Dist: ZZZ - Consumo: VVV**", donde XXX podrá ser encendido o apagado; YYY será el nivel del depósito expresado en litros y con dos decimales; ZZZZ será la distancia recorrida expresada en kilómetros y con dos decimales, y VVV el consumo expresado en litros y con dos decimales. Algunos ejemplos de la salida de ***toString*** podrían ser:
 1. motor: apagado - Depósito: 0,00 - Dist: 0,00 - Consumo: 0,00
 2. motor: arrancado - Depósito: 4,98 - Dist: 0,00 - Consumo: 0,02
 3. motor: arrancado - Depósito: 4,23 - Dist: 15,00 - Consumo: 0,77
 4. motor: apagado - Depósito: 4,23 - Dist: 0,00 - Consumo: 0,00
 5. motor: arrancado - Depósito: 4,68 - Dist: 10,00 - Consumo: 0,32
 9. Documenta apropiadamente el código con **comentarios javadoc** para poder generar la documentación apropiadamente:
 1. **documentación a nivel de clase**. Debes incluir un buen comentario para documentar la clase en general;
 2. **documentación a nivel de métodos**, incluyendo, por cada método:
 1. **descripción de los atributos públicos**. Uso de `@value` para incluir el valor de las constantes;
 2. **descripción del método. Descripción corta y descripción larga**;
 3. **descripción de los parámetros** (`@param`);
 4. descripción de las posibles excepciones que puede lanzar (`@throws`). En nuestro caso serán **IllegalArgumentException** o **IllegalStateException**. En algunos casos ambas.
 10. Se proporciona un **programa principal** para probar el funcionamiento de la clase **Vehículo** con tres variables objeto v1, v2 y v3 de esa clase. El programa está casi completo. Tan solo falta añadir el código necesario para probar el objeto v3 sobre el que habrá que realizar las siguientes acciones:
 1. Crear el objeto vehículo **v3 con el constructor sin parámetros**;
 2. Repostar 20 litros de combustible en el vehículo v3;
 3. Arrancar el vehículo v3;
 4. Recorrer un trayecto de 100 km con el vehículo v3;
 5. Dejar el motor del vehículo v3 encendido, es decir, no apagarlo.
 11. Fíjate bien en cómo se pueden realizar esas acciones observando cómo se ha hecho previamente con v1 y v2, enviando a la pantalla los mensajes apropiados y gestionando las excepciones que sean necesarias de una manera apropiada. Debes "proteger" esas acciones con bloques try - catch de la misma manera que se ha hecho previamente con las operaciones sobre v1 y v2.
- Es **obligatorio** que utilices esa clase principal que se te proporciona. De esta manera todos tendrás que amoldarlos a las interfaces definidas para la clase **Vehículo**. Es decir, que vuestra clase **Vehículo** deberá encajar en este programa principal como si fuera una pieza más de un **puzzle** como ya hemos tenido que hacer en otras ocasiones. La clase principal la puedes descargar de la sección de recursos necesarios. En esta clase se utilizan métodos de la clase **ES** para la lectura de datos por teclado. Por tanto, también deberás incorporarla en tu proyecto.

II.2.- Información de interés

Recursos necesarios y recomendaciones

- ✓ Recuerda que **es obligatorio utilizar la clase principal** que se te proporciona como programa principal de tu proyecto con su método main(): [Clase Principal](#) (java - 15.18 KB).
- ✓ La clase ES para recepción de información por teclado y escritura en pantalla: [Clase ES](#) (java - 17.27 KB)
- ✓ Las excepciones que hay que lanzar desde los métodos de la clase Vehiculo (y capturar en el programa principal) son IllegalStateException y IllegalArgumentException.
- ✓ Para que te sea más fácil probar si tu programa funciona correctamente, te proporcionamos un ejemplo de ejecución del programa con diversos **casos de prueba**. Deberías probar tu programa al menos con todos esos casos, más otros que se te puedan ocurrir a ti, pues los profesores vamos a probar esos **como mínimo**:
 - ⇒ Archivo de texto (.txt) con la salida del programa al probar todos esos casos: [Ejecución de casos de prueba](#) (txt - 6.96 KB)
- ✓ También te proporcionamos una muestra de cómo podría quedar tu **documentación javadoc**. Tus descripciones y comentarios no tienen porqué 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): [Ejemplo de Javadoc](#) (zip - 64.8 KB).

II.3.- Propuesta de solución

Aquí tienes una propuesta de solución al ejercicio: [solución clase Vehiculo](#) (zip - 27.3 KB).