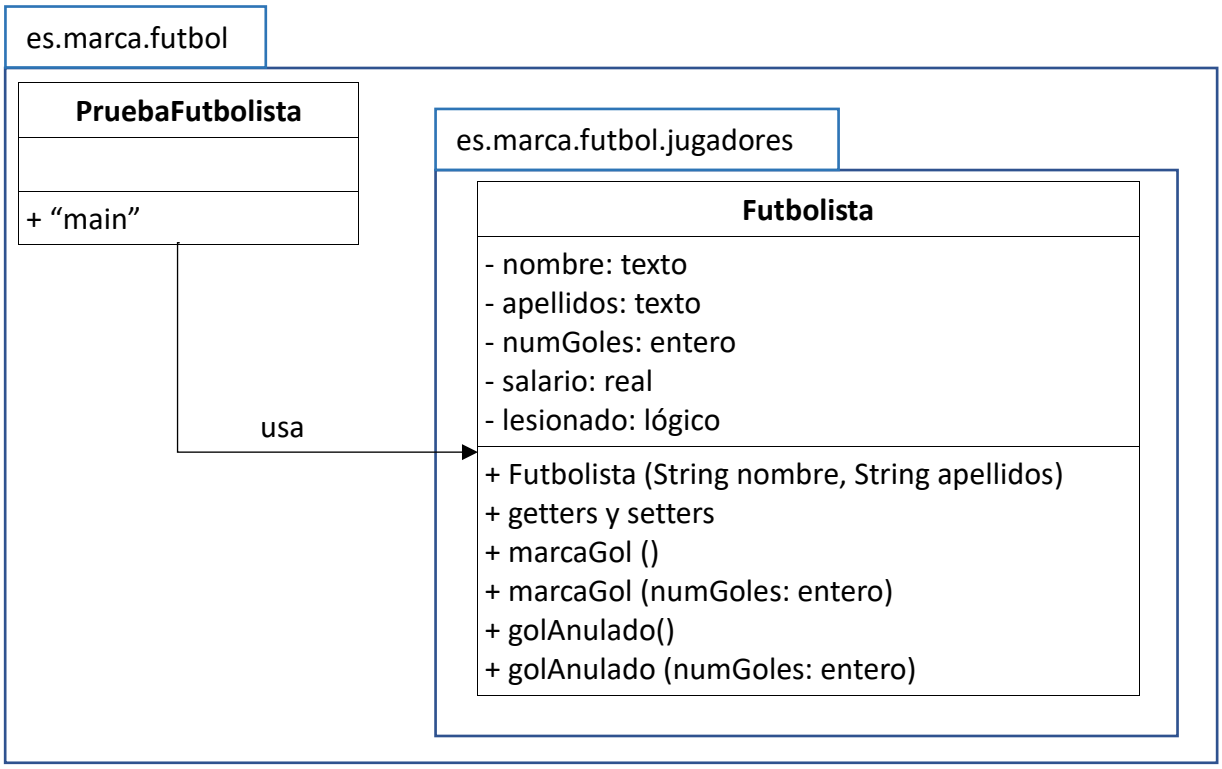
CUESTIONES Y EJERCICIOS – ENCAPSULAMIENTO, CONSTRUCTORES Y SOBRECARGA

1. Teclea la siguiente clase de modo aplicando las técnicas de encapsulamiento y la sobrecarga de constructores:

Ordenador
- marca: texto - modelo: texto - anioFabricacion: entero - encendido: lógico
+ Ordenador() + Ordenador(marca: texto, modelo: texto) + Ordenador(marca: texto, modelo: texto, anio: entero) + imprimeEstado() + getters y setters

   A continuación, crea una clase *PruebaOrdenador* con un método *main* en la que crees tres objetos *Ordenador* utilizando un constructor distinto cada vez. Una vez creados los tres objetos debes utilizar el método *imprimeEstado()* de cada uno de ellos que mostrará el valor de las propiedades del objeto.

2. Escribe la clase *Futbolista* y pruébala con una clase *PruebaFutbolista* siguiendo el siguiente diagrama:



Debes atender, además, a las siguientes restricciones:

- **marcaGol()** incrementará en uno la cuenta de goles del futbolista.
- **marcaGol (numGoles)** incrementará la cuenta de goles del futbolista añadiendo los que se reciben como parámetro.
- **golAnulado ()** si el número de goles actual es 0 se mostrará el mensaje: "Error: no se puede anular el gol", en caso contrario se restará un gol a la cuenta de goles del futbolista.
- **golAnulado (numGoles)** solo realizará la operación de decrementar el número de goles restando los que se reciben como parámetro si el resultado de la resta es mayor o igual que cero. En caso contrario, se mostrará el mensaje: "Error: no se pueden anular *numGoles* goles"
- La clase **PruebaFutbolista** debe crear un objeto Futbolista y probar los métodos sobrecargados.

3. Vamos a rescatar la clase Vehiculo que hicimos en el tema 1. Está en el repositorio de esta unidad didáctica con el nombre: U1.Extra.Vehiculo.zip. A continuación, vamos a crear una clase Persona que pueda ser propietario de hasta 3 vehículos, atendiendo al siguiente modelo:

Persona
- nombre: texto - dni: texto - coches: Vehiculo[3] - numCochesActual: entero
+ Persona () + Persona (nombre: texto, dni: texto) + comprarCoche(coche: Vehículo) + arrancarCoche(coche: Vehículo) + apagarCoche(coche: Vehículo) + recorrerDistancia(coche: Vehículo, numKm: real) + reponerCombustible(coche: Vehículo, numlitros: real) + getters de todas las propiedades + setters solo de *nombre* y *dni*

Además, debes tener en cuenta que:

- El método *comprarCoche* añade el vehículo que recibe como parámetro a la siguiente posición libre del array. En caso de que el array ya no tenga más posiciones libres entonces debe mostrar el mensaje: “Ya no puedes tener más coches”.
- El método *arrancarCoche* y *apagarCoche* utilizan los métodos correspondientes del objeto Vehículo que reciben como parámetro.
- El método *recorrerDistancia* utiliza el método correspondiente del Vehículo que recibe como parámetro pasándole a su vez el número de kilómetros recorridos a este método.
- El método *reponerCombustible* utiliza el método correspondiente del Vehículo que recibe como parámetro pasándole a su vez el número de litros repuestos a este método.

Por último, debe crear una clase PruebaPersonaCoche que pruebe todos los métodos de Persona utilizando 2 coches en primer lugar. Posteriormente se deben comprar 2 coches más, de modo que al intentar comprar el 4º coche salga el mensaje de error correspondiente.

4. Observa la clase del diagrama UML.

Mamifero
- nombre: texto - anioNacimiento: entero
+ getters y setters + haceRuido()

   Escríbela y en el método *haceRuido* simplemente imprime en pantalla *"No sé qué ruido hago. Puedo ser muchas cosas"*

   A continuación, extiende la clase anterior creando dos subclases llamadas *Felino* y *Canido.*

   Extiende la clase Felino, creando dos subclases llamadas *Gato* y *Tigre.*

   Extiende la clase Canido, creando la clase *Perro.*

   Sobrescribe el método *haceRuido* en las siguientes clases para que se comporte del siguiente modo:

   - En la clase Gato debe imprimirse "Miauuuu"
   - En la clase Tigre debe imprimirse "Roooooaooaarrr"
   - En la clase Perro debe imprimirse "Guau"

   Por último, escribe una clase *PruebaMamiferos* que cree un objeto de cada una de las clases creadas y pruebe su método *haceRuido*.

   ***OJO**: en este caso **NO** nos interesa que los métodos haceRuido llamen al método de su clase padre con super.haceRuido();*

5. Realiza un diagrama UML en el que se modele una herencia entre clases de dispositivos de un ordenador sabiendo que:

   - Todos los dispositivos se pueden encender/apagar y saber si están encendidos o no.
   - Todos los dispositivos tienen una marca y un modelo y unos métodos que permiten acceder y modificar estos valores.
   - El procesador es un dispositivo que puede sumar, restar, multiplicar y dividir dos números enteros, devolviendo el resultado correspondiente. Además, llevará la cuenta de cuántas operaciones ha realizado desde que se encendió por última vez.
   - La memoria RAM es un dispositivo capaz de guardar hasta 100 números enteros. Además, permite acceder a cada uno de estos números según su posición (0…99) para modificar su valor (escribir en memoria) o bien obtener dicho valor (leer de memoria). Además, llevará la cuenta de cuántas operaciones de lectura y cuántas operaciones de escritura se han realizado desde que se encendió por última vez.
   - La tarjeta de vídeo es un dispositivo capaz de mostrar en la pantalla números enteros y textos. Además, llevará la cuenta de cuántas operaciones de mostrar un número entero y cuántas operaciones de mostrar un texto se han realizado desde que se encendió por última vez.

   Una vez tengas le diagrama con la jerarquía de clases y las propiedades/métodos que componen cada clase, codifícalo.

   Por último, haz una clase *PruebaDispositivos* que cree un procesador, una memoria y una tarjeta de vídeo y pruebe cada uno de los métodos que has desarrollado.

6. Retoma el ejercicio anterior y sobrescribe el método toString() en todas las clases, teniendo en cuenta que las subclases deben hacer uso del método toString de su superclase y, además, imprimir el estado de sus propiedades. Por ejemplo, esta podría ser la cadena de texto devuelta por el método *toString()* de un objeto de la clase Procesador.

   Dispositivo: marca=Intel, modelo=i5, encendido=true. Procesador: operaciones=6.

7. Retoma el ejercicio anterior y añade el siguiente método abstracto a la clase Dispositivo:

   public abstract void resetContadores();

   Este método deberá ser implementado por las subclases y su comportamiento consiste en poner a cero todos los contadores que haya en la subclase.

8. Imagina que ahora decidimos reorganizar clase Dispositivo añadiendo las siguientes subclases:

   - Entrada: que representaría a los dispositivos de solo entrada de datos en un ordenador (teclado, ratón…)
   - Salida: que representaría a los dispositivos de solo salida de datos en un ordenador (altavoces, pantalla…)
   - EntradaSalida: que representaría a los dispositivos que permiten tanto la entrada como la salida de datos en un ordenador (impresora…)

   Supongamos que estas subclases no tienen métodos propios, escribe cómo declararías cada una de las tres clases. A continuación, responde a las siguientes preguntas:

   - ¿Tendría sentido crear un objeto de la clase Entrada con un new Entrada()?
   - Ahora queremos crear una clase Impresora extendiéndola desde la clase EntradaSalida ¿qué métodos heredaría con código? ¿Y cuáles sin código?
   - ¿Se podría crear la clase Impresora anterior, pero extendiendo de las clases Entrada y Salida simultáneamente?
   - ¿Tendría sentido decir que la clase Entrada es abstracta y final a la vez? Explícalo razonadamente.

9. Retoma el código del ejercicio 7 y añade el siguiente constructor a la clase Dispositivo:

   public Dispositivo(String marca, String modelo)

   Este constructor debe asignar la marca y el modelo a las propiedades respectiva. Sin añadir ningún constructor más a la clase Dispositivo, toma las medidas que creas oportunas para arreglar los errores que se producen en las subclases.

10. Partimos de nuevo del código del ejercicio anterior para realizar una nueva clase llamada *TestDispositivos* que se dedica a encender y apagar dispositivos 100 veces para comprobar su correcto funcionamiento.

    Esta clase contendrá un método *main* que utilizará un array de 5 dispositivos de distintas clases (por ejemplo: 2 procesadores, 2 memorias y 1 tarjeta de vídeo). Debes escribir el código que permite encender y apagar 100 veces los 5 dispositivos del array.

    OJO: no vale hacer 5 bucles, uno por dispositivo… Esta solución no sería "escalable" ¿Qué pasa si mañana el array es de 50 dispositivos?

11. Observa las siguientes interfaces:

```
public interface IArticulo {
    String getNombre();
    void setNombre(String nombre);
    double getPrecio();
    void setPrecio(double precio);

    // Sobrescribimos el método toString de Object para que
    // devuelva una cadena con el formato:
    // Artículo: nombre. Precio: precio €
    @Override
    String toString();
}

public interface ICarroCompras {
    // Añade un artículo más al carro, añadiéndolo a la
    // colección (array) existente.
    void meteEnCarro(Articulo art);

    // Recorre el carro mostrando los artículos almacenados
    void muestraArticulosCarro();

    // Devuelve el número de artículos guardados en el carro
    int getNumArticulosActual();

    // Elimina los artículos del carro
    void vaciarCarro();

    // Devuelve el sumatorio de los precios de los artículos
    // guardados en el carro
    double calculaImporteCarro();
}
```

Las clases Articulo y CarroCompras que he usado para realizar el reto de Jamazon de esta unidad implementan ambas interfaces respectivamente.

Fíjate en la potencia expresiva que tienen las interfaces. Con solo dos ficheritos .java que puedo facilitar a un equipo de desarrolladores o mi alumnado estoy siendo capaz de expresar lo siguiente:

- Quiero un diseño con, al menos, 2 clases.
- Quiero que cada clase tenga al menos el conjunto de métodos que especifica la interfaz.
- Quiero que los métodos se llamen exactamente así y tengan esos parámetros de entrada y salida.
- Estoy dejando intuir las propiedades que debe tener cada clase.
- Y además también se deja entrever cómo se relacionan ambas clases.

Además, el compilador se convierte en nuestro aliado porque dará error si alguna de las clases no cumple correctamente con la interfaz que implementa.

Si todavía no has conseguido superar este reto quizás sea el momento de intentarlo de nuevo.

12. Dada la siguiente interfaz:

```
public interface InterfazDocumento {
	String getTitulo();
	void setTitulo(String titulo);
	String getContenido();
	void setContenido(String contenido);
	int getNumPaginasImpresion();
	void setNumPaginasImpresion(int numPaginas);
}
```

Crea una clase que implemente *InterfazDocumento*. A continuación, crea una clase *PruebaDocumento* que tenga un método *main* que cree un objeto de la clase que has creado que pruebe todos los métodos que has creado.

13. Continúa el ejercicio anterior, añadiendo la siguiente interfaz:

```
public interface InterfazImpresora {
	String getMarca();
	String getModelo();
	void encender();
	void apagar();
	void cargarBandeja(int numFolios);
	void imprimir(InterfazDocumento documento);
	String getEstado();
}
```

Crea una clase *Impresora* que implemente la *InterfazImpresora* teniendo en cuenta que:

- Cuando se llame al método *imprimir* se debe imprimir en pantalla el siguiente mensaje: "Se está imprimiendo el documento *título* que ocupa *xx* páginas".
- Sin embargo, en el caso de que el número de folios en la bandeja no sea suficiente para imprimir un documento completo entonces se emitirá el mensaje: "Error: no hay suficiente papel para imprimir el documento *título".*
- El método *getEstado* devolverá el texto "OK" siempre que la impresora esté encendida y se haya podido imprimir un documento completo. Devolverá "Falta papel" cuando se intente imprimir un documento que ocupa más del número de páginas disponible. Si la impresora está recién encendida se devolverá "OK" y si está apagada se devolverá "Fuera de servicio".
- El método *cargarBandeja* añade el número de folios que se reciben como parámetro a los que hubiera en la bandeja.

A continuación, crea una clase *PruebaImpresora* que tenga un método *main* que cree un objeto de tipo *Impresora* que tenga como marca "HP" y modelo "Laserjet 2000". Se debe cargar con 5 folios. Por otro lado, debes crear dos documentos, uno que ocupe 4 folios y otro que ocupe 3. Debes intentar imprimir ambos documentos y ofrecer al usuario los mensajes que correspondan.

14. Dada la siguiente interfaz:

```
public interface InterfazFichero {
	String getNombre();
	void setNombre(String nombre);
	void setCodificacion(String codificacion);
	String getCodificacion();
	void abrir();
	void cerrar();
	void borraContenido();
	void agregaContenido(String contenido);
	String getContenido();
	int getTamanioEnBytes();
}
```

Crea una clase que implemente *InterfazFichero* sabiendo que:

- Los ficheros deben crearse con un nombre antes de poder ser abiertos.
- Para poder manipular un fichero debe estar abierto.
- El método *agregaContenido* añade el nuevo contenido al final del contenido que hubiera.
- La codificación del fichero puede ser "UTF-8" o "UTF-16". Se aplicará "UTF-8" como codificación por defecto y también, en el caso de que el usuario especifique alguna otra codificación desconocida.

- El método *getTamanioEnBytes* devolverá el número de bytes que ocupa el contenido del fichero sabiendo que cada carácter en UTF-8 ocupa 1 byte y que cada carácter en UTF-16 ocupa 2 bytes.

A continuación, crea una clase *PruebaFichero* que tenga un método *main* que cree un objeto de la clase que has creado que pruebe todos los métodos que has creado.

15. Dada la siguiente interfaz:

```
public interface InterfazMensajeMail {
	void setAsunto(String asunto);
	String getAsunto();
	void setContenido(String contenido);
	String getContenido();
	void setRemitente(String direccionEmail);
	String getRemitente();
	void setDestinatario(String direccionEmail);
	String getDestinatario();
	void setFicheroAdjunto(InterfazFichero fichero);
	void eliminaFicheroAdjunto();
	InterfazFichero getFicheroAdjunto();
	void envia();
}
```

Crea una clase que implemente *InterfazMensajeMail* sabiendo que:

- Los métodos *setRemitente* y *setDestinatario* deben comprobar que el parámetro contiene el carácter @ y que la longitud de la cadena es mayor o igual a 5 caracteres. En caso contrario no se guardarán los valores en las propiedades correspondientes.
- El método *envia* debe comprobar que el existe la dirección del remitente y del destinatario, emitiendo un mensaje de error en caso contrario. Si el asunto o el contenido están vacíos entonces se imprimirá un mensaje de advertencia tipo “OJO: estás enviando un mensaje sin asunto/contenido”. A continuación, se enviará el mensaje y se imprimirá en la pantalla alguno de los siguientes mensajes según corresponda:
  - “Mensaje enviado de remitente@xxxx a destinatario@yyyy con asunto xxxxxxxxxx”
  - “Mensaje enviado de remitente@xxxx a destinatario@yyyy con asunto xxxxxxxxxx y con un fichero adjunto de xx bytes”

A continuación, crea una clase *PruebaMensajeMail* que tenga un método *main* que cree los objetos necesarios para probar todos los métodos de la clase que has creado.

16. Descarga el documento “Tabla de modificadores.docx” de la Moodle y complétalo, explicando qué hace cada modificador sobre cada elemento y o bien escribiendo “NO APLICA” si un modificador no se puede utilizar sobre cierto elemento del lenguaje.

17. Imagina una pila de libros muy pesados, “tipo enciclopedia”, de modo que solo podría coger uno cada vez porque pesan demasiado. Este tipo de pila (stack, en inglés) se comporta como sigue:

    - Si hay espacio disponible en la pila (si no hemos tocado el techo de la habitación) entonces puedo poner un libro en la cima de la pila. A esto le llamamos *apilar* (push, en inglés)
    - Si quiero quitar un libro de la pila, tiene que ser el de la cima, no vale quitar alguno de en medio. A esto se le llama *desapilar (*pop, en inglés).
    - Si no hay libros en la pila y quiero desapilar se provocaría un error.
    - Si ya hemos llegado al techo y quiero apilar otro libro también se provocaría un error (stackoverflow, en inglés “desbordamiento de pila”)

    Esta estructura de datos la utiliza el compilador para gestionar el paso de los parámetros a los métodos de un objeto en la memoria.

    Nuestro Libro lo modelaremos así:

Libro
- titulo: texto - precio: real
Libro (titulo: texto, precio: real) + getters y setters + toString() devuelve texto

    Bueno, pues ahora tienes que desarrollar una clase que implemente la siguiente interfaz de modo que, utilizando un array como colección de elementos, pueda comportarse como una pila de libros. Y la interfaz a implementar será:

```
public interface IPilaLibros {
    void apilar(Libro l);
    Libro desapilar();
    Libro verCima();
    int getNumElementos();
    void vaciar();
}
```

    La clase que implemente la interfaz deberá tener un constructor que reciba como parámetro el número máximo de elementos que podrá contener el array.

    Por último, crea una clase llamada PruebaPilaLibros que pruebe la clase que has creado.