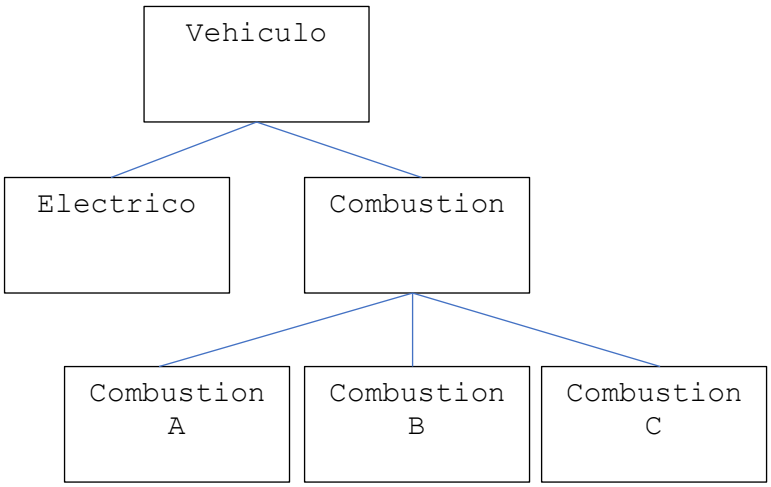
La Dirección General de Tráfico (DGT) quiere actualizar su sistema de control impuestos. Para ello, va a definir una jerarquía de clases que permita diferenciar los tipos de vehículos en eléctricos y de combustión y, dentro de éstos últimos, los de tipo A, B o C; siguiendo el esquema:



Ejercicio 1. (4 puntos)

Codificar las clases `Vehiculo`, `Electrico`, `Combustion`, `CombustionA`, `CombustionB` y `CombustionC`, teniendo en cuenta que:

- Todo vehículo queda identificado de forma única por su matrícula (tipo `String`).
- Desde el punto de vista fiscal los vehículos eléctricos pagan un 5%, los de combustión tipo A un 10% en función de su cilindrada (tipo `double`, medida en litros), los de tipo B un 15% en función de su cilindrada y los de tipo C un 20% en función de su cilindrada.
- La clase `Vehículo` deberá tener un método `impuesto` que devuelve el porcentaje fiscal aplicado al vehículo (tipo `double`).

Ejercicio 2. (2 puntos)

Se quiere mejorar la jerarquía de clases antes definida con un control de errores sobre las matrículas de los vehículos. De modo que sólo se consideren matrículas válidas aquellas cuyo patrón este compuesto por 4 números y 3 letras. A la hora de crear un vehículo se deberá verificar la matrícula y en caso de que ésta no cumpla con lo indicado lanzar la excepción `MatriculaException`.

Para ello, se pide modificar las clases de la jerarquía anteriormente definida que se consideren necesarias para incluir el tratamiento de errores especificado y crear la clase `MatriculaException` para el tratamiento de la excepción, que consistirá en la definición de un mensaje del tipo: "`Matrícula [DSH1234] no válida...`".

Ejercicio 3. (4 puntos)

Codificar la clase `GestorImpuestos` que permitirá gestionar los impuestos vinculados a los vehículos de todos los contribuyentes (número no determinado) que tengan al menos un vehículo asociado. Un contribuyente es identificado de forma única por su DNI (tipo `String`). Se deberán codificar, al menos, siguientes métodos de la clase:

- **`public void`** `altaVehiculo(String dni, Vehiculo vehiculo)`

  Este método da de alta un `vehiculo` vinculándolo al contribuyente identificado por `dni`. En caso de que el vehículo ya esté dado de alta se generará una excepción que consistirá en la definición de un mensaje del tipo: "`Vehículo con matrícula [DSH1234] dado de alta previamente…`".

- **`public boolean`** `consultarMatricula(String matricula)`

  Este método indica si la `matricula` pasada por parámetro se corresponde con algún `vehiculo` dado de alta (`true`) o si no se encuentra dada de alta (`false`).

- **`public double`** `obtenerImpuestos(String dni)`

  Este método recupera el impuesto del contribuyente vinculado al `dni`, como la suma de los impuestos de todos los vehículos que tiene asociados. En el caso de que el `dni` pasado por parámetro no corresponda a ningún contribuyente existente el método devolverá `0.0`.

- **`public ¿?`** `obtenerImpuestos()`

  Este método recupera una estructura de datos (**`¿?`**) que vincula cada contribuyente con el impuesto total que se le aplica. Se deberá decidir qué estructura de datos a devolver es la más adecuada teniendo en cuenta la naturaleza del problema.

INFORMACIÓN ADICIONAL:

java.util
Interface Map<K,V>

- **boolean containsKey(Object key).** Returns true if this map contains a mapping for the specified key.
- **boolean containsValue(Object value).** Returns true if this map maps one or more keys to the specified value.
- **Set<K> keySet().** Returns a Set view of the keys contained in this map.
- **Collection<V> values().** Returns a Collection view of the values contained in this map.
- **V put(K key, V value).** Associates the specified value with the specified key in this map (optional operation).
- **V remove(Object key).** Removes the mapping for a key from this map if it is present (optional operation).
- **int size().** Returns the number of key-value mappings in this map.

java.util
Interface List<E>

- **boolean add(E e).** Appends the specified element to the end of this list (optional operation).
- **void add(int index, E element).** Inserts the specified element at the specified position in this list (optional operation).
- **boolean contains(Object o).** Returns true if this list contains the specified element.
- **E get(int index).** Returns the element at the specified position in this list.
- **int indexOf(Object o).** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
- **E remove(int index).** Removes the element at the specified position in this list (optional operation).
- **boolean remove(Object o).** Removes the first occurrence of the specified element from this list, if it is present (optional operation).
- **int size().** Returns the number of elements in this list.

java.util
Interface Set<E>

- **boolean add(E e).** Adds the specified element to this set if it is not already present (optional operation).
- **boolean contains(Object o).** Returns true if this set contains the specified element.
- **int size().** Returns the number of elements in this set (its cardinality).
- **Iterator<E> iterator()**. Returns an iterator over the elements in this set.

Examen de Programación Orientada a Objetos (Plan 2014)

24 de junio de 2022

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 1. (4 puntos)

```
public abstract class Vehiculo {
    private final String matricula;

    public Vehiculo(String matricula){
        this.matricula = matricula.toUpperCase();
    }

    public abstract double impuesto();
}

public class Electrico extends Vehiculo{
    public Electrico(String matricula) {
        super(matricula);
    }

    @Override
    public double impuesto() {
        return 5.0;
    }
}

public abstract class Combustion extends Vehiculo{
    private final double cilindrada;

    public Combustion(String matricula, double cilindrada){
        super(matricula);
        this.cilindrada = cilindrada;
    }

    protected double getCilindrada() {
        return cilindrada;
    }
}

public class CombustionA extends Combustion{
    public CombustionA(String matricula, double cilindrada){
        super(matricula, cilindrada);
    }

    @Override
    public double impuesto() {
        return 10.0 * getCilindrada();
    }
}
```

```java
public class CombustionB extends Combustion{
    public CombustionB(String matricula, double cilindrada){
        super(matricula, cilindrada);
    }

    @Override
    public double impuesto() {
        return 15.0 * getCilindrada();
    }
}


public class CombustionC extends Combustion{
    public CombustionC(String matricula, double cilindrada){
        super(matricula, cilindrada);
    }

    @Override
    public double impuesto() {
        return 20.0 * getCilindrada();
    }
}
```

Corrección: **1 punto por la clase** `Vehiculo`
1 punto por la clase `Electrico`
1 punto por la clase `Combustion`
1 punto por las clases `CombustionA, CombustionB` **y** `CombustionC`

Examen de Programación Orientada a Objetos (Plan 2014) — **24 de junio de 2022**

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 2. (2 puntos)

```
public abstract class Vehiculo {
    private final String matricula;

    public Vehiculo(String matricula) throws MatriculaException {
        matricula = matricula.toUpperCase();
        if (matricula.matches("^[0-9]{4}[A-Z]{3}$")) {
            this.matricula = matricula;
        }else{
            throw new MatriculaException(matricula);
        }
    }

    public abstract double impuesto();
}

public class MatriculaException extends Exception{
    public MatriculaException(String matricula) {
        super("Matrícula ["+matricula+"] no válida...");
    }
}
```

En el resto de las clases definidas es necesario añadir el lanzamiento de la excepción en la firma de su constructor: `throws MatriculaException`

Corrección: **1 punto por la clase** `Vehiculo`
0,5 puntos por la clase `MatriculaException`
0,5 puntos por la modificación del resto de clases

```java
/* codificación alternativa sin expresiones regulares */

public abstract class Vehiculo {
    private final String matricula;

    public Vehiculo(String matricula) throws MatriculaException {
        matricula = matricula.toUpperCase();
        if (validarLongitud(matricula) && validarNumeros(matricula) &&
            validarLetras(matricula)) {
            this.matricula = matricula;
        }else{
            throw new MatriculaException(matricula);
        }
    }

    private boolean validarLongitud(String matricula){
        return (matricula.length() == 7);
    }

    private boolean validarNumeros(String matricula){
        for (int i=0; i<4; i++) {
            char numero = matricula.charAt(i);
            // Si no está entre 0 y 9
            if (!(numero >= '0' && numero <= '9')) {
                return false;
            }
        }
        return true;
    }

    private boolean validarLetras(String matricula){
        for (int i=4; i<7; i++) {
            char letra = matricula.charAt(i);
            // Si no está entre A y Z
            if (!(letra >= 'A' && letra <= 'Z')) {
                return false;
            }
        }
        return true;
    }

    public abstract double impuesto();
}
```

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

Ejercicio 3. (4 puntos)

```
import java.util.ArrayList;
import java.util.HashMap;

public class GestorImpuestos {
    private HashMap<String, ArrayList<Vehiculo>> contribuyentes;

    public GestorImpuestos() {
        contribuyentes = new HashMap<>();
    }

    public void altaVehiculo(String dni, Vehiculo vehiculo)
           throws VehiculoExistenteException {
        if (consultarMatricula(vehiculo.getMatricula()))
            throw new VehiculoExistenteException(vehiculo);
        if(!contribuyentes.containsKey(dni)){
            contribuyentes.put(dni, new ArrayList<>());
        }
        contribuyentes.get(dni).add(vehiculo);
    }

    public boolean consultarMatricula(String matricula) {
        for (String dni : contribuyentes.keySet()) {
            for(Vehiculo vehiculo : contribuyentes.get(dni)){
                if(vehiculo.getMatricula().equals(matricula)){
                    return true;
                }
            }
        }
        return false;
    }

    public double obtenerImpuestos(String dni){
        double impuestos = 0.0;
        if(contribuyentes.containsKey(dni)) {
            for (Vehiculo v : contribuyentes.get(dni)) {
                impuestos += v.impuesto();
            }
        }
        return impuestos;
    }

    public HashMap<String, Double> obtenerImpuestos(){
        HashMap<String, Double> impuestos = new HashMap<>();
        for (String dni : contribuyentes.keySet()) {
            impuestos.put(dni, obtenerImpuestos(dni));
        }
        return impuestos;
    }
```

```
        private class VehiculoExistenteException extends Exception {
            public VehiculoExistenteException(Vehiculo vehiculo) {
                super("Vehículo con matrícula ["+vehiculo.getMatricula()+"]
                       dado de alta previamente...");
                }
        }

}
```

Corrección: **1 punto por la correcta elección de estructuras de datos**
1,5 puntos para el método `altaVehiculo` con su correspondiente excepción
0,5 puntos para cada uno de los restantes métodos