Proyecto: Aseguradora de vehículos

Para la versión V3 del proyecto se habrán de cumplir los requisitos descritos a continuación:

Requisitos V3

Partiendo de la versión V2 del proyecto:

Importa en el proyecto la librería **AseguradoraUtils.jar**

Hay dos formas de importar una librería descargada en un proyecto Maven, dependiendo de si quieres instalar la librería en el repositorio local o si se quiere usarla directamente en el proyecto.

Opción 1: Instalar la librería en el repositorio local de Maven

Es la forma más recomendable si se quiere usar la librería en varios proyectos.

1. Abre una terminal y ejecuta el siguiente comando para instalar la librería en tu repositorio local (.m2):

```
mvn install:install-file -Dfile=/ruta/a/tu/libreria.jar -DgroupId=com.miempresa
-DartifactId=milib -Dversion=1.0 -Dpackaging=jar
```

2. Luego, en el pom.xml, se añade la dependencia:

```
<dependencies>
    <dependency>
        <groupId>com.aseguradora.utils</groupId>
        <artifactId>aseguradora-utils</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
```

NOTA: si Maven (mvn) no se reconoce, primero habrá que descargarlo para poder aplicar esta solución. Una vez descargado (https://maven.apache.org/download.cgi) habrá que agregar la ruta al PATH.

1. Pulsa Win + R, escribe sysdm.cpl y presiona Enter.
2. Ve a la pestaña **"Opciones avanzadas"** y haz clic en **"Variables de entorno"**.
3. En **"Variables del sistema"**, busca la variable Path y edítala.

Si nos ubicamos en el directorio en el que está la librería, un ejemplo de comando funcional podría ser:

```
mvn install:install-file -Dfile=AseguradoraUtils.jar -DgroupId=com.aseguradora.utils
-DartifactId=aseguradora-utils -Dversion=1.0 -Dpackaging=jar
```

Opción 2: Usar la librería desde un directorio dentro del proyecto

Si prefieres no instalar la librería en Maven, puedes colocarla en un directorio dentro del proyecto y hacer que Maven la reconozca.

1. Crea una carpeta dentro del proyecto, por ejemplo, libs, y coloca el .jar ahí.
2. Modifica el **pom.xml** para agregar la librería como dependencia local:

```
<dependency>
   <groupId>com.miempresa</groupId>
   <artifactId=milib</artifactId>
   <version>1.0</version>
   <scope>system</scope>
   <systemPath>${project.basedir}/libs/milib.jar</systemPath>
</dependency>
```

Nota: El scope **system** no es recomendable para producción, pero funciona si solo necesitas usar la librería sin instalarla.

Resumen:
Si la librería es reutilizable en otros proyectos, es mejor instalarla en el repositorio local (Opción 1).
Si es solo para este proyecto y no quieres instalarla, puedes usar la dependencia system (Opción 2), aunque no es la mejor práctica.

Nota: Puede ser necesario indicar en nuestro IDE la versión de maven que vamos a utilizar si se ha instalado la dependencia por consola.

En esta librería tendrás 4 clases disponibles (todas en el paquete com.aseguradora.utils):

Marca
Tiene dos atributos: nombre (String) de la marca y modelos (List<Modelo>).
Además de los getter y los setter, tiene:
addModelo(Modelo m): añade un modelo m a la lista de modelos
removeModelo(Modelo m): elimina el modelo m de la lista de modelos
listaModelos(): devuelve una lista de nombres de Modelo de la Marca (List<String>)
getModelo(String m) devuelve el Modelo de la marca que se llama m, si existe.

Modelo
Tiene 4 atributos: nombre (string) del modelo y los precios base de seguro a terceros, terceros ampliado y todo riesgo: precioTERC(double), precioTAMP(double) y precioTRIE(double).

Tarifa
TIene los atributos: marca (String), modelo (String), año de matriculación: anyo (int), monedaPrecios (String) (por defecto €), y los precios “finales” para la combinación de marca, modelo y ano de matriculación: precioTERC, precioTAMP, precioTRIE.

SoporteVehiculos
Una clase que se instancia mediante SoporteVehiculos.getInstance(). Tiene dos atributos, que se inicializan al instanciar la clase por primera vez:
marcasCoches (Map<String, Marca>)
marchasMotos (Map<String, Modelo>)
Y están disponibles sus getters.
Tiene los métodos:
listaMarcasCoches: devuelve una List<String> con las marcas válidas de coches.
listaMarcasMotos: devuelve una List<String> con las marcas válidas de motos
listaMarcas() devuelve una lista de marcas válidas (de coches o de motos).
getMarcasCochesList: devuelve un List<Marca> de coches
getMarcasMotosList: devuelve un List<Marca> de motos
getMarcas(): devuelve List<Marca> de coches y Motos
getMarcaByName(String marca) devuelve una Marca por el nombre dado, si existe.
esMarcaValida(String marca) devuelve si el nombre de una marca es válido
esModeloValido(String marca, String modelo): devuelve si un nombre de marca y modelo dados son una combinación válida
calcularTarifa(String marca, String modelo, int anyo) devuelve un objeto de tipo Tarifa, basado en el nombre de marca, modelo y año de matriculación del vehículo.
listaModelos(String marca): devuelve una Lista<String> de modelos basado en el nombre de una marca de entrada.
multiplicadorCP(String CP): devuelve un multiplicador (double) basado en el CP (en el que se supone que pasará más tiempo el vehículo).

Una vez visto todo esto

Modifica la clase **Vehiculo** de la siguiente forma:
Cambia el atributo marca de tipo String a tipo **Marca**
Cambia el atributo modelo de tipo String a tipo **Modelo**

Deja los constructores existentes (en los que estos dos parámetros se pasaban como String), pero crea otros constructores en los que se pasen como Marca y Modelo.
En los que se pasen como String, tendrá que validarse que la Marca y Modelo sean válidos. Si no lo son, en vez de crearse el objeto, tendrá que lanzarse una IllegalArgumentException (hay que declarar el constructor con throws IllegalArgumentException).
Además, deberá verificarse que la fecha de matriculación es anterior o igual al día en curso y que la matrícula del vehículo es válida (utilizar métodos de la V2).

También habrá que cambiar la clase **Persona** para que se verifique que el NIF es válido.

También hay que cambiar la clase **Conductor** para verificar que es mayor de edad.

Se creará una clase **Provincia**

- nombre -> String
- codigo ->String

Todos los modelos han de contar con los atributos privados que se detallan, al igual que definir al menos un **constructor** vacío, copia y completo.
Han de tener definidos los método **toString**, **equals** y **hashCode** (buscad información de este si no habéis llegado al ejercicio en que se necesita).

Se cambiará provincia de String a Provincia en **Direccion**.

En utilidades, hay que crear una clase **UtilidadesDireccion**, que deberá tener el método

public static boolean esCPValido(String cp)

que devolverá que es válido si los dos primeros caracteres son dígitos del 01 al 52 y los tres últimos son dígitos (es decir, desde el 01001 hasta el 52999).

También se creará un Map<String, Provincia> (estático) que contendrá todas las provincias válidas con su código, la clave será el nombre de la provincia:

Código	Provincia
01	Álava
02	Albacete
03	Alicante
04	Almería
05	Ávila
06	Badajoz
07	Islas Baleares
08	Barcelona

09 Burgos
10 Cáceres
11 Cádiz
12 Castellón
13 Ciudad Real
14 Córdoba
15 A Coruña
16 Cuenca
17 Girona
18 Granada
19 Guadalajara
20 Guipúzcoa
21 Huelva
22 Huesca
23 Jaén
24 León
25 Lleida
26 La Rioja
27 Lugo
28 Madrid
29 Málaga
30 Murcia
31 Navarra
32 Ourense
33 Asturias
34 Palencia
35 Las Palmas
36 Pontevedra
37 Salamanca
38 Santa Cruz de Tenerife
39 Cantabria
40 Segovia
41 Sevilla
42 Soria
43 Tarragona
44 Teruel
45 Toledo
46 Valencia
47 Valladolid
48 Vizcaya
49 Zamora
50 Zaragoza
51 Ceuta
52 Melilla

En el paquete src > test > java se ha de crear una clase main llamada **pruebaV3**
Aquí se probarán los nuevos métodos y cambios en clases: creación de Personas con las validaciones indicadas (validación de NIF, edad de conductor, que el CP sea válido y que

coincidan los dos primeros dígitos con el código de la provincia de la dirección), creación de Vehículos (Coche o Moto) con las validaciones indicadas (Marca, Modelo, matrícula, año de matriculación, etc.).
Crea Coches, Motos, Personas y Conductores (o intentes crearlos, si devuelven excepciones captúralas para no parar el programa).

Una vez hecho todo esto, intenta generar Tarificaciones para vehículos, tanto válidas como erróneas, y mostrarlas por consola.

Excepciones

Para capturar excepciones en Java, se utiliza el bloque

try{
	//código que puede elevar la excepción
}**catch**(Exception e){
	//código para tratar la captura de la excepción
}**finally**{
	//código que hay que ejecutar se dé o no excepción, como cierre de ficheros
}

El bloque finally es opcional, y puede haber varios bloques catch (yendo siempre de excepciones concretas o diferentes a más generales).

Si un código puede generar una excepción pero no la trata (no la captura con un bloque try-catch, o la captura para tratar algo, pero la lanza de nuevo con un **throw**, el método que puede generarla deber indicar que puede lanzar excepciones con la palabra reservada **throws** seguridad de los tipos de excepciones que puede lanzar.

Ejemplo:

```
public class EjemploExcepcion {
   public static void metodoPeligroso() throws Exception {
       try {
           int resultado = 10 / 0; // Esto genera una ArithmeticException
       } catch (ArithmeticException e) {
           throw new Exception("Error en metodoPeligroso: división por cero", e);
       }
   }

   public static void main(String[] args) {
       try {
           metodoPeligroso();
       } catch (Exception e) {
           System.out.println("Excepción capturada en main: " + e.getMessage());
           e.printStackTrace();
       }
   }
}
```