CUESTIONES Y EJERCICIOS

1. Vamos a crear una clase *Teclado* que tendrá el siguiente diagrama UML:

Teclado
- Scanner: sc
+ Teclado() + leeEntero() devuelve entero + leeEntero (int minimo, int maximo) devuelve entero + leeEnteroMax (int maximo) devuelve entero + leeEnteroMin (int mínimo) devuelve entero

Los métodos de esta clase deben modelar los siguientes comportamientos:

- **Teclado()** será un constructor que se limitará a inicializar la propiedad *sc* con un objeto de la clase Scanner.
- **LeerEntero()** devolverá el entero que haya puesto el usuario por teclado. En el caso de que el usuario escriba algo que no pueda ser convertido a un entero (por ejemplo: "nueve", 54.12…) entonces este método debe manejar la excepción, indicándole al usuario que se ha equivocado y permitiéndole introducir de nuevo el número tantas veces como sea necesario.
- **LeerEntero (int mínimo, int máximo)** hace lo mismo que el método anterior pero además valida que el entero que se haya introducido esté comprendido entre los valores mínimo y máximo que se reciben como parámetros. En el caso de que el valor se salga de este rango permitido debe mostrar un mensaje de error tipo: "Debes teclear un entero comprendido entre *mínimo* y *máximo"*
- **LeerEnteroMax (int máximo)** hace lo mismo que el método leerEntero pero además valida que el entero que se haya introducido sea menor o igual que el entero que se recibe como parámetro. En el caso de que el valor se salga de este rango permitido debe mostrar un mensaje de error tipo: "Debes teclear un entero menor o igual que *máximo"*
- **LeerEnteroMin (int mínimo)** hace lo mismo que el método leerEntero pero además valida que el entero que se haya introducido sea mayor o igual que el entero que se recibe como parámetro. En el caso de que el valor se salga de este rango permitido debe mostrar un mensaje de error tipo: "Debes teclear un entero mayor o igual que *mínimo"*

Crea una clase PruebaTeclado con un método main que cree un objeto de la clase teclado y pruebe los métodos tanto con valores correctos como forzando los posibles fallos que se puedan dar.

2. Vamos a retocar la clase anterior para añadir un conjunto de métodos nuevos que se basan en el uso de un método muy utilizado para hacer la lectura de números del teclado en distintos formatos.

   El método estático Integer.parseInt(String s) analiza la cadena *s* que recibe como parámetro y devuelve un *int* que se corresponde con la conversión de la cadena a número entero, siempre que se pueda. Por ejemplo:

   ```
   int valor = Integer.parseInt(“-123”);
   ```

   Esto asignaría el número -123 a la variable *valor*

   En el caso de que la cadena no pudiera ser convertida a un entero entonces se lanzaría una *NumberFormatException*.

   Además, existe una variante muy útil de este método que es:

   ```
   Integer.parseInt(String s, int radix)
   ```

   Se comporta igual que el anterior, pero permite especificar la base (radix) en la que queremos entender el número. Por ejemplo:

   ```
   int valor = Integer.parseInt(“100”, 2);
           // En base 2 (binario)
   ```

   Esto asignaría el número 4 a la variable *valor*

   Se pide retocar la clase Teclado para incorporar los siguientes métodos. En el caso de que se produzcan excepciones deben capturarse y permitir que el usuario vuelva a introducir el número tantas veces como sea necesario:

   - **LeerBinario()** que devolverá el entero correspondiente a la cadena de números binarios (de 0 a 1) que haya introducido el usuario por teclado.
   - **LeerOctal ()** que devolverá el entero correspondiente a la cadena de números en base 8 (de 0 a 7) que haya introducido el usuario por teclado.
   - **LeerHex ()** que devolverá el entero correspondiente a la cadena de números en base hexadecimal (de 0 a 9 y de A a F) que haya introducido el usuario por teclado.

   Retoca la clase PruebaTeclado para que se añadan las pruebas de los nuevos métodos.

3. Una inmobiliaria gestiona apartamentos en alquiler. Queremos modelar una clase *Apartamento* de modo que pueda guardarse su dirección postal, el número de habitaciones, el número de camas de las que dispone, si está actualmente ocupado o libre.

   La clase debe tener un constructor que reciba todos los parámetros de entrada para establecer el valor de todas las propiedades. Asimismo, tendrá un método ocupar() y otro liberar().

   La clase debe "defenderse" ante malos usos lanzando excepciones en los siguientes casos:

   - Si el constructor se utiliza de forma que se intente guardar un *null* en la propiedad de la dirección se debe lanzar una NullPointerException acompañada de un mensaje descriptivo.
   - Asimismo, si el constructor se llama con parámetros absurdos como que el apartamento tiene -9 camas o 100000 habitaciones se debe lanzar una IllegalArgumentException acompañada de un mensaje descriptivo.
   - Si se llama al método liberar() y el apartamento ya está libre se debe lanzar una IllegalStateException acompañada de un mensaje descriptivo.
   - Si se llama al método ocupar() y el apartamento ya estaba ocupado se debe lanzar una IllegalStateException acompañada de un mensaje descriptivo.

   Por último, crea una clase PruebaApartamento con un método main que pruebe la clase Apartamento lanzando las 4 excepciones descritas. Estas excepciones deberán ser capturadas en el main y se deberá mostrar tanto el nombre de la clase que las genera (toString) como el mensaje asociado a la excepción (getMessage).

4. Vamos a crear una clase CuentaCorriente que se caracteriza por su saldo actual y un entero largo que identifique a la cuenta. Además, guardará también una cadena de 4 dígitos que represente el PIN de la tarjeta de débito que tenga asociada.

   La clase debe tener un constructor que reciba todos los parámetros de entrada para establecer el valor de todas las propiedades. En el caso de que alguno de los parámetros del constructor sea incoherente o incumpla con el formato especificado se debe lanzar una IllegalArgumentException.

   Asimismo, tendrá los siguientes métodos:

   - *double consultaSaldo()* que devolverá el saldo actual
   - *boolean intentaAccesoConPin(String pin)* se devolverá *true* si el PIN que se recibe como parámetro coincide con el que se puso en el constructor al crear el objeto.
   - long getNumeroCuenta() que devuelve el número de cuenta.
   - *void ingresa (double importe)* sumará el importe que se recibe como parámetro al saldo de la cuenta. Lanzará una IllegalArgumentException si el importe es cero o menor que cero.
   - *void abona (double importe)* restará el importe que se recibe como parámetro del saldo de la cuenta. Lanzará una IllegalArgumentException si el importe es cero o menor que cero.

   Por último, crea una clase PruebaCuentaCorriente con un método *main* que cree un objeto de la clase CuentaCorriente y pruebe todos sus métodos, así como que se lanzan las respectivas excepciones cuando corresponde.

5. Tomando como modelo el código de *U5.P3.FicheroThrowsCheckedException* vamos a crear una clase *EscrituraFichero* que tenga la misma estructura que la clase Fichero del código de partida. Sin embargo, quitaremos el método *imprimeContenido* y añadiremos uno nuevo llamado *public void escribeTexto (String linea) throws IOException.*

   También dejaremos de usar las referencias a las clases FileReader y BufferedReader y las cambiaremos por FileWriter y PrintWriter respectivamente.

   De estas dos nuevas clases debes saber lo siguiente:

   - La clase FileWriter permite crear objetos que saben escribir caracteres a un fichero.
   - La clase PrintWriter permite escribir líneas completas a un fichero de forma cómoda para el programador, permitiendo el uso del método *println(…)*

   Todos los métodos deben lanzar sus excepciones, no capturan nada.

   Por último, debes realizar una clase de prueba llamada *PruebaEscrituraFichero* con un main haga lo siguiente:

```
Dime una frase y te la escribo a un fichero:
Con cien cañones por banda, viento en popa a toda vela

Dime el nombre del fichero donde la guardo:
poema.txt
Frase guardada correctamente
```

   El código del main debe estar dentro de un bloque *try* de forma que se capture una IOException globalmente. En caso de que se produzca la excepción debe mostrarse el mensaje "Algo falló al abrir o escribir la frase en el fichero"

6. Añade al ejercicio anterior un método nuevo con la siguiente firma:

```
public void escribeTexto(List<String> texto) throws IOException
```

   que escriba una lista de cadenas al fichero.

7. Con lo que has aprendido sobre el bloque *finally* añade un nuevo método a la clase del ejercicio anterior con la firma:

```
public void abreEscribeCierra(List<String> texto)
```

   de modo que realice la operación y se capturen las posibles excepciones que ocurran. Si ocurre una excepción se mostrará el mensaje *"Se produjo un error al intentar abrir o escribir en el fichero".*

   En cualquier caso se debe garantizar que el método cierra todos los manejadores del fichero antes de que devuelva el control a la clase cliente.

8. Vamos a crear una clase Banco que permita una "conexión con el exterior" para atender peticiones de pago con tarjeta de crédito. Esta clase almacenará las siguientes propiedades:
    - Un mapa llamado *cuentasCorrientes* que usará como clave el número de cuenta y como valor un objeto de la clase CuentaCorriente.
    - Un mapa llamado *cuentasConectadas* que usará como clave el token de conexion y como valor un objeto de la clase CuentaCorriente.

    Y los siguientes métodos:

    - *public int abreConexion (long numeroCuenta, String pin) throws CuentaOPinIncorrectoException* este método devolverá un "token de conexión" que será un número aleatorio entre 0 y 999999 y que servirá para identificar la cuenta una vez se haya comprobado que el PIN es correcto.

      Se debe buscar la cuenta indicada en el mapa *cuentasCorrientes* y comprobar que el pin recibido coincide con el almacenado en la cuenta. Si todo es correcto se generará el "token de conexión" y se almacenará en *cuentasConectadas* usando como clave el token y como valor el objeto CuentaCorriente. Por último, se devolverá dicho token.

      Si la cuenta no existe o el pin no es correcto se debe lanzar una CuentaOPinIncorrectoException.
    - *public void realizaPago (int tokenCuenta, double importe) throws TokenIncorrectoException, SaldoInsuficienteException* este método debe comprobar que el token recibido es uno de los tokens válidos almacenados en *cuentasConectadas*. Si no es un token válido se lanzará una TokenIncorrectoException.

      Si el token es válido se comparará el importe recibido con el saldo de la cuenta, si se puede realizar el pago se restará el importe del saldo de la cuenta. En caso contrario se lanzará una SaldoInsuficienteException.
    - *public void cierraConexion (int tokenCuenta) throws TokenIncorrectoException* este método debe borrar el token recibido del mapa de las *cuentasConectadas*. En el caso de que no exista el token recibido se debe lanzar la excepción.
    - Un constructor sin parámetros que inicializará los dos mapas. Además, al mapa de *cuentasCorrientes* se deben crear y añadir 4 objetos CuentaCorriente que usaremos para nuestras pruebas.

    Por último, crea una clase PruebaBanco con un método *main* que cree un objeto de la clase Banco y pruebe todos sus métodos simulando una compra con una tarjeta de débito siguiendo los pasos *abreConexion -> realizaPago -> cierraConexion*.