EJERCICIO 1

Crea un programa que solicite números enteros al usuario y los meta en una lista. Cuando indique el número -1, parará de meter números. Al final, imprime la lista.

Una vez que tengas el programa, ¿cómo podrías asegurarte de que el scanner se cierre siempre, aunque haya algún error que pare la ejecución?

EJERCICIO 2

En el programa anterior, si te fijas, cuando en lugar de un número el usuario introduce letras, salta un error y el programa termina. Captura dicha excepción para que el programa no termine, sino que simplemente muestre un error al usuario indicando que no puede introducir letras.

PISTA: después de mostrar el mensaje tendrás que hacer un nextLine() para consumir el salto de línea.

EJERCICIO 3

Amplia el programa anterior creando una clase que sea SacoNumeros que tenga como atributo una lista de Integer. Haz que tu aplicación use esta clase en lugar de tenerlo todo "apiñado".

La clase SacoNumeros tendrá que tener un método addNumero() que te permita ir añadiendo dichos números y otro método toString() que imprima todos los números.

Una vez que lo tengas modificado, añade otro método que sea getNumero() que recibe un Integer que será la posición y devuelve el número que está en dicha posición.

Haz que tu programa pregunte al usuario qué números quiere ver solicitando posiciones y se los muestre. Hasta que el usuario indique la posición -1. En tal caso, dejarás de preguntarle.

EJERCICIO 4

En el programa anterior, cuando te han indicado una posición que no existe en la lista, salta una excepción y el programa termina. Captura dicha excepción para que en tal caso el número devuelto sea null.

EJERCICIO 5

Añade a tu clase un método que sea division() que devuelva un BigDecimal con el resultado de dividir el primer número de la lista entre el segundo, y el resultado entre el tercero, y el resultado entre el cuarto, y así sucesivamente hasta terminar la lista. El resultado de cada división parcial estará siempre redondeado a 2 decimales con HALF_UP.

Si te encuentras con algún valor que sea 0, la división fallará. Captura la excepción para que en ese caso el resultado sea igual a cero.

Prueba el nuevo método llamando desde tu programa tras lo hecho en el Ejercicio 4.

EJERCICIO 6

Construye la clase Persona que tenga como atributos el género (String) y la altura (BigDecimal). Crea método toString() y sus métodos get() y set() teniendo en cuenta que:

- El método setGenero() sólo admitirá recibir por parámetro el valor H o M. Si recibe cualquier otra cosa, tendrá que lanzar una excepción nueva llamada "ParametroIncorrectoException" indicando el detalle del error.
- El método setAltura() solo admitirá un BigDecimal comprendido entre 0 y 3, ambos excluidos. Si recibe cualquier otra cosa, tendrá que lanzar también una ParametroIncorrectoException indicando el detalle del error.

Haz una aplicación para permitir al usuario crear Personas indicando los datos por consola. Cuando lo haga, imprime los datos. Controla las excepciones que pueden originarse e imprime el mensaje de error correspondiente.

EJERCICIO 7

Construye Sociedad que tenga como atributo un conjunto (un Set) de personas. Añade los siguientes métodos:

- addPersona() que recibe por parámetro un String con el género y un BigDecimal con la altura. El método tendrá que crear una nueva persona y añadirla a la lista. Captura las excepciones que puedan originarse al crear la persona, muestra un mensaje de error si salta alguna y vuelve a lanzarlas.
- calcularAlturaMediaHombres() y calcularAlturaMediaMujeres() que devuelva un BigDecimal con la altura media de las personas que hay en la lista según su género. Si no hubiera ninguna persona del género indicado, lanzará una excepción llamada "ListaVaciaException" indicando el detalle del error.
- calcularAlturaMedia() que hace lo mismo que los anteriores, pero sin filtrar por género
- ¿No crees que los tres métodos anteriores son muy parecidos? ¿Estás repitiendo código? Piensa en un buen diseño…

Crea una aplicación que solicite al usuario en un bucle datos de personas hasta que nos diga que terminemos. Después, muestra la altura media de hombres, mujeres y total.

Controla todas las excepciones para que el programa no termine con errores, sino que el usuario esté informado siempre de lo que ocurre.

EJERCICIO 8

Prueba a cambiar en el ejercicio 6 y 7 para que tus dos excepciones hereden de Exception o de RuntimeException. Mira cómo se modifica el comportamiento a la hora de programar.

EJERCICIO 9

Añade un método a la clase Sociedad que sea borrarGenero() que reciba por parámetro un String con el género que se quiere borrar. Se deben eliminar del conjunto todas las personas de ese género. Si el género recibido no es ni H ni M, lanzará una excepción "ParametroIncorrectoException". Prueba el método desde tu aplicación.

EJERCICIO 10

Recupera el ejercicio 46 del tema 4. Lanza una FechaIncorrectaException en todos aquellos métodos (incluido constructor) donde pudiera crearse una fecha con parámetros incorrectos.

Prueba el resultado con un pequeño programa.

EJERCICIO 11 (avanzado)

Crea una clase que sea ArbolGenealogico. Esta clase permitirá almacenar el árbol genealógico de cualquier persona. Asumiremos que sólo hay un emparejamiento y que no es posible tener hijos sin estar emparejado.

La clase Nodo tendrá estos atributos:

- "progenitor1" y "progenitor2" que serán de tipo Nodo
- "conyuge" que será de tipo Nodo
- "nombre" que será de tipo String
- "fechaNacimiento" y "fechaDeceso" que serán LocalDate
- "hijos" que será una lista de Nodo

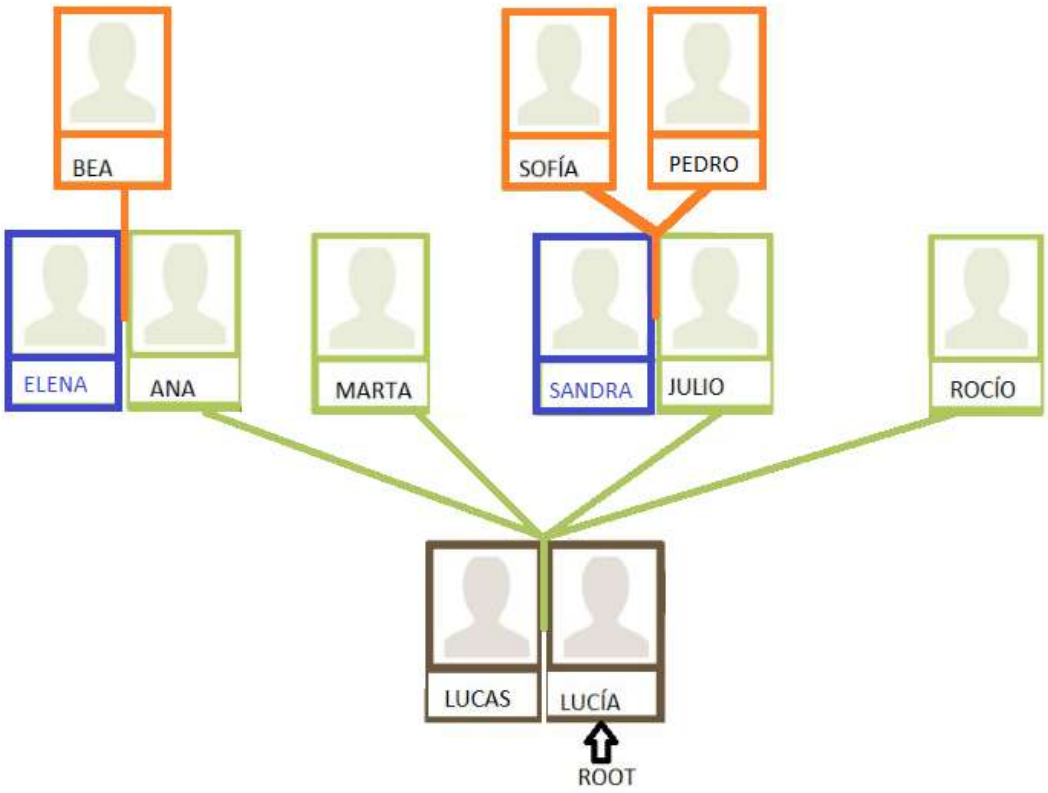
La clase Nodo tendrá un constructor para inicializar nombre y fechaNacimiento. Tendrá métodos get y set para todos sus atributos. Además, tendrá estos métodos:

- establecerConyuge() → Recibe un nodo y lo establecerá como cónyuge propio si no tiene ya uno asignado. Si tuviera uno asignado, lanzará ConyugeException con un mensaje.
- addHijo() → Recibe un nombre y una fecha de nacimiento. Si el nodo no tiene cónyuge lanzará ConyugeException con un mensaje. En caso contrario, creará un nuevo nodo con los datos recibidos, completará el de progenitor1 y progenitor2, y añadirá el hijo a la lista. El hijo tendrá que añadirse también a la lista de hijos del cónyuge.
- getHijo() → Devuelve un nodo con el hijo que tenga el nombre que se reciba por parámetro. Si no existe, lanzará SinHijosException con un mensaje.
- getPrimogenito() → Devuelve el hijo mayor. Si no tuviera hijos, lanzará SinHijosException con un mensaje
- getBenjamin() → Devuelve el hijo menor. Si no tuviera hijos, lanzará SinHijosException con un mensaje.
- getHermanos() → Devuelve una lista con todos los nodos hermanos. Si el nodo no tuviera progenitores, lanzará SinPadresException con un mensaje.
- estaVivo() → devuelve un booleano
- tieneHijos() → devuelve un booleano

- tienePadresVivos() → devuelve un booleano. Si no tuviera progenitores, lanzará SinPadresException con un mensaje.

La clase ArbolGenealogico tendrá como atributo un Nodo al que llamaremos root que no tendrá padres. Sobre este root es sobre el que se irán añadiendo hijos. Por lo que únicamente tendrá un constructor que reciba los datos del root (Nombre y fecha de nacimiento) para inicializarlo y get/set del root con el que se trabajará desde fuera.

Crea un programa que haciendo uso de las clases anteriores permita crear este árbol:



Puedes inventar las fechas de nacimiento y decesos.

Prueba los métodos creados para obtener información.

¿Se te ocurre qué otros métodos útiles podríamos añadir?