Revisión y repaso para el examen. Algunas cuestiones que debes recordar

1. Usa la generación automática y luego retoca si hace falta en los métodos:
   - *Getters y setters*
   - *toString*
   - *equals*

2. La clase *String* tiene algunos métodos que debes recordar:
   - *string.toUpperCase()*. Pasa toda la cadena a mayúscula.
   - *string. substring(posición inicio, posición fin (sin incluir)*. Ejemplo: "Esternocleoidomastoideo". *substring(4,7)* = "rno" (4, 5 y 6). (Se empieza a contar en 0).
   - *string.trim()* elimina los blancos en ambos extremos.

3. Conviene llevar en el método *parse* el número de línea para incluirlo en el error.

4. Crear los *set/get* con la mínima visibilidad posible (set privados o *protected* si son de superclase y hacen falta en subclases), *get* depende de si es necesario verlos desde fuera.

5. En la ordenación, el orden ascendente o descendente depende de cómo realices el método *compareTo* o el método *compare*.

```
            return this.getName().compareTo(c.getName());// ASCENDENTE
//          return c.getName().compareTo(this.getName());   DESCENDENTE
```

6. Ocurre igual con el método compare

```
        return c1.getAmountOwned() - c2.getAmountOwned();   // ASCENDENTE
//      return c2.getAmountOwned() - c1.getAmountOwned();   // DESCENDENTE
```

7. En comparadores, si quieres comparar
   - Cadenas: usa *cadena.compareTo(cadena2)*
   - Números enteros: usa *numero1 – numero2*
   - Números *double*: primero haces un *cast* a los números para transformarlos en objetos de tipo *Double*, que tienen el método comparar.
     - *((Double) numeroDouble1).compareTo((Double) numero2).*

8. Para crear una clase de pruebas pulsa botón derecho sobre la clase que quieras probar, pulsa *new/JUnit testCase* y cambia el nombre de la carpeta *src* por *test*. Te creará la clase en un paquete con igual nombre al original. Si no existe ya crea el paquete. Acuérdate que debes usar *JUnit 4*

9. En los test si hay que comparar valores de tipo *double*, el *assert* debe llevar tres parámetros. Ejemplo *assertEquals(esperado, real, 0.1)*; . Te darás cuenta si al escribirlo el compilador tacha la palabra *assertEquals* (le falta el último parámetro).

10. Cuando quieras en un test comparar dos objetos, ten en cuenta que debe estar redefinido el método *equals* en la clase de esos objetos, de lo contrario fallará el test si los objetos no son el mismo (igualdad

de identidad) o hay dos referencias apuntando al mismo objeto. Otra opción es comparar el *toString()* de cada uno. En cualquier caso, el *equals* lo puedes generar automáticamente.
assertEquals(o1.toString(), o2.toString()); dará true si ambos tienen los mismos datos.
assertEquals(o1,o2); dará true si tienen los mismos datos y está implementado el *equals* en la clase de estos objetos. Si no está implementada sólo da *true* si *o1* y *o2* tienen la misma referencia.

11. Cuando tengas varios test, crea la clase *testAll* (menú *testSuite*) e incluye todas las clases de test.

12. Implementa *toString()* siempre que necesites que se vayan a imprimir los datos (es muy conveniente en las clases del modelo, las que contienen los datos).
    Implementa *equals(Object o)* siempre que se necesite saber si dos objetos son iguales. (por ejemplo, para saber si un objeto ya está o no en una lista). Se puede generar automáticamente, luego se repasa por si no coincide con lo que pide el programa (comparará todos los campos y a veces no es preciso, depende de lo que nos indique el ejercicio).

13. Implementa *compareTo(T objeto)* si hay que hacer ordenación. Normalmente se usa primero el *compareTo* de la interfaz *Comparable<T>* (siendo T el tipo de los objetos que se desean comparar) y si hay otras ordenaciones adicionales se crean clases **comparadores** implementando la interfaz *Comparator<T>* y se implementa el método *compare*.

14. Implementa *serialize()* siempre que se vaya a guardar datos en un fichero.

15. Subdivide las operaciones en otras más pequeñas e impleméntalas en métodos aparte. Por ejemplo puedes usar un método *checkXX* o *assertXX* , siempre que tengas que realizar algún control. Usa también métodos *handleXX* para manejar las excepciones. Utiliza esta técnica siempre que veas que una operación es algo compleja. Divídela en pasos y usa un método para cada paso.

16. Realiza las operaciones en la clase donde están los datos. De esta manera se reduce el acoplamiento.
    - Por ejemplo, si hay que serializar los objetos de una lista, para cada elemento podemos sacar todos sus atributos y crear un *String* con ellos o podemos pedir al objeto que devuelva en un *String* todos sus atributos. Esta opción siempre es mejor.
    - Si tienes que saber el valor de un atributo booleano, puedes pedir el atributo y comprobar si es true (por ejemplo if (person.getState () == MARIED). Otra opción es llamar a if (person.isMaried()). Esta opción es mejor porque no obtenemos el atributo (la implementación del dato) sino el dato en sí. Si se cambia la implementación, con que siga existiendo la interfaz es suficiente. Las clases estarán menos acopladas.

17. Recoge todas las posibles excepciones de forma que no dejes que el programa rompa, sino que acabe ordenadamente, si tiene que acabar, y muestre un mensaje al usuario como mínimo. Resulta interesante también añadir más información del error en un fichero log. Recuerda que suele haber como mínimo dos posibles manejadores, el *handleUserError* (que se le dice al usuario que arregle el problema) y el *handleSystemError* (que se le dice al usuario que avise al servicio técnico).

18. Los errores de programación o del sistema se propagan hasta la clase de arranque (hasta el run) porque no es posible recuperarse de ellos. Simplemente se manejan mostrando al usuario un mensaje para que avise al servicio técnico, y se envía la información del error a un log. *IOException* se ha decidido considerarla como error del sistema salvo el caso de *FileNotFoundException* que puede ser un error que el usuario puede resolverlo.

19. Los errores que se consideren y que son de la lógica del funcionamiento de la aplicación lanzan excepciones propias. Son errores para el usuario. Su manejo implica informar al usuario y su recogida depende.

- Si se quiere acabar el programa se propagan hacia atrás hasta el run. Se informa al usuario por si puede resolverlo, en ocasiones si podrá. Por ejemplo, *FileNotFound* si no hay *UserInterface* y no se puede pedir de nuevo el nombre del fichero, el usuario de la aplicación tendrá que colocar el fichero en la carpeta adecuada y con el nombre adecuado, tras lo cual volver a ejecutar el programa.
- Si se quiere que se pueda seguir ejecutando otra operación se propaga la excepción hasta la operación de la clase gestora donde se produjo. Por ejemplo, añadir un objeto al sistema, si el objeto ya existe se informa al usuario y se continúa (si así se requiere en el enunciado) con otras operaciones. Si hubiera una clase tipo *UserInterface* con un menú de opciones, se propagan hasta aquí para que el usuario pueda seleccionar otra opción del menú.
- Si se quiere que continúe haciendo algo dentro de la operación donde se produjo, se recogerá en el punto donde haya algún bucle que permita continuar con otro elemento. Por ejemplo, en la operación *cargarLista* en el sistema, si un objeto ya existe y no se quieren repetir se lanzará excepción, pero se recoge en el bucle para que sigan cargándose otros elementos de la lista.

20. Cuando se crea un flujo (stream) se debe siempre cerrar incluyendo un try{ … }finally que empieza después de crear el flujo y acaba después de realizar las operaciones de uso del flujo (leer o escribir).