Método equals:

➔ Nos brinda la manera de cómo comparar objetos a partir de una solución lógica.

➔ **cuando NO:**

❖ Cada instancia de la clase es propiamente única.

❖ No hay necesidad para la clase de proveer una igualdad lógica.

❖ Una superclase ya implementó este método y el mismo es apropiado para la clase.

❖ La clase es privada y estamos seguros que el método equals nunca será invocado.

➔ **cuando SI:**

❖ Cuando una clase tiene una igualdad lógica que va más allá de la identidad del objeto. Y una superclase no haya implementado este método

¿Que establece su contrato?:

El método equals nos va a dar una certera relación de equivalencias entre las distintas
instancias.

➔ Reflexivo: Para cada valor de X no nulo. X.equals(X) es true.

➔ Simétrico: Para cada valor de X e Y no nulo. X.equals(Y) es true si y sólo si Y.equals(X) es true.

➔ Transitivo: Para cada valor de X, Y, Z no nulo. Si X.equals(Y) es true Y Y.equals(Z)
es true Entonces X.equals(Z) debe ser true.

➔ Consistente: Para cada valor de X e Y. Múltiples invocaciones de X.equals(Y) debe
retornar consistentemente true o consistentemente false. Siempre que no se modifique la información utilizada en ambos. Para cada valor de X no nulo. X.equals(null) debe retornar false

Receta para un método equals de calidad:

1. Usar el operador == para chequear si el argumento es una referencia a este objeto(la ocupación de memoria). Si lo es retorna true.
2. Usar el operador instanceof para chequear si el argumento posee el tipo correcto(si
estamos comparando dos instancias de la misma clase). Si no, retorna false.
3. Castear el argumento al tipo correcto. Como el casteo se realiza después de un instanceof está garantizado que será satisfactorio.
4. Por cada atributo "significante" (cada atributo que queramos analizar dentro de la clase) en la clase chequear si ese atributo es igual al atributo correspondiente de este objeto. Si todos estos son satisfactorios, retorna true. Caso contrario false.

Para los atributos primitivos cuyo tipo no es float ni double usar el operador == para comparar.

Para objetos, llamar al método equals recursivamente. Para atributos float usar el método estático compare de la clase Float. Float.compare(float1, float2) Para atributos double usar el método estático compare de la clase Double. Double.compare(double1, double2)

Método hashCode ¿Que establece su contrato? :

➔ Cuando el método hashCode es invocado en un objeto repetidas veces debe

retornar el mismo valor consistentemente.
➔ Si dos objetos son iguales de acuerdo al método equals, entonces hashCode debe
retornar el mismo valor para ambos.
➔ Si dos objetos son distintos de acuerdo al método equals, no es necesario que hashCode produzca un valor distinto. Sin embargo esto puede mejorar la performance de una hash table.
Receta para un buen método hashCode:
1. Declarar una variable entera llamada resultado e inicializarla con el valor de hash code del primer atributo significante en nuestro objeto.
2. Para cada atributo remanente en nuestro objeto hacer lo siguiente:
A. Si el atributo es de tipo primitivo computar Type.hashCode(atributo). Donde Type es el Tipo correspondiente a la primitiva.
B. Si el atributo es un objeto y esa clase hace uso de equals de manera recursiva para sus atributos. Usar hashcode recursivamente en el atributo. Si el valor del atributo es null usar 0.
C. Si el atributo es un array, tratarlo como si cada elemento fuera un atributo separado. Esto significa calcular un hashcode para cada elemento en el array y combinar los valores. Si el array no tiene elementos significantes usar una constante diferente de 0. Si TODOS los elementos son importante usar Arrays.hashCode

Método hashCode:

Siempre se debe hacer Override de hashCode cuando se hace
Override de equals.

Ejemplo método hashCode

```
public class Telephone {
private short areaCode;
private short prefix;
private short lineNum;
...
public int hashCode() {
int result = Short.hashCode(areaCode);
result = 31 * result + Short.hashCode(prefix);
result = 31 * result + Short.hashCode(lineNum);
return result;
}
}
```

3. Combinar el resultado de esta forma: resultado = 31 * resultado + atributo
Interfaz Comparable:
➔ **public interface Comparable { int compareTo(T t); }**
➔ Al implementar esta interfaz en una clase estamos indicando que las instancias de
dicha clase poseen un orden natural. Es decir que la interfaz nos va a indicar que las
instancias de dicha clase poseen un orden natural

El contrato de este método especifica que debe devolver lo siguiente:

➔ Entero negativo si el objeto es menor al especificado por parámetro.
➔ Cero si el objeto es igual al especificado por parámetro.
➔ Entero positivo si el objeto es mayor al especificado por parámetro.

Interfaz Map

➔ Nos permite representar una estructura de datos para almacenar pares "clave-valor".
➔ Para una clave sólo tenemos un valor.
➔ Map tienen implementada por debajo toda la teoría de las estructuras de datos de árboles, por lo tanto permiten añadir, eliminar y modificar elementos de forma transparente.
➔ La clave funciona como un identificador único y no se admiten claves duplicadas.

HashMap

➔ Un HashMap es una colección de objetos, como los arrays, pero estos no tienen orden definido
➔ Cada objeto se identifica mediante algún identificador (nuestra clave) y conviene que sea inmutable (final), de modo que no cambie en tiempo de ejecución.
➔ El nombre Hash hace referencia a una técnica de organización de archivos llamada hashing o "dispersión" en el cual se almacenan registros en una dirección del archivo que es generada por una función que se aplica a la clave del mismo.
➔ Los elementos que se insertan en un HashMap no tendrán un orden específico.
➔ Permite una clave null.

Principio de Hashing

➔ Se utiliza el método hashCode() para encontrar el bucket correspondiente.
➔ Se utiliza el método equals() para buscar el valor correspondiente a la clave dada.
➔ Dos objetos idénticos tendrán el mismo identificador retornado por el método hashCode().

HashMap vs. Hashtable:

Importante :
"Si dos objetos son iguales usando equals(), entonces la invocación a hashCode() de ambos objetos debe retornar el mismo valor."

➔ La principal diferencia entre uno y otro es la sincronización interna del objeto. Para aplicaciones multihilos es preferible elegir Hashtable sobre HashMap, que no tiene sincronización.

➔ HashMap es mejor en cuanto a performance.
➔ Hashtable no admite valores nulos en ninguna de sus partes, mientras que HashMap
sí.

LinkedHashMap:

➔ Similar a HashMap (ya que la implementa, es una clase que la extiende) pero con la
diferencia que mantiene una lista doblemente vinculada, además del array de baldes y cada uno de sus contenidos de valores.
➔ La lista doblemente vinculada define el orden de iteración de los elementos
➔ Esto nos permite tener el conocimiento de en qué orden fueron agregados cada una
de estas claves valores a un HashMap

TreeMap:

➔ Implementado mediante un árbol binario y permite tener un mapa ordenado.
➔ Cuando iteramos un objeto TreeMap los objetos son extraídos en forma ascendente
según sus keys. Nos organiza dentro de nuestro has cada posición de los elementos por su tipo.
➔ **TreeMap**
no sabe cómo ordenar la colección cuando se utiliza una key creada por el programador, sólo lo hace si trabajamos con objetos tipo String, Integer, etc.

Queue:
➔ Representa al tipo Cola, que es una lista en la que sus elementos se introducen únicamente por un extremo (fin de la cola) y se remueven por el extremo contrario (principio de la cola). Dependiendo su tamaño

Interfaz Set:

➔ Modela los conjuntos de la matemática y sus propiedades.
➔ Set hereda los métodos de Collection y agrega sus propias restricciones para prohibir el duplicado de elementos.
➔ Si tenemos un objeto que tienen las mismas características (equals) y el mismo hashcode que los objetos que ya se encuentran en el Set → No se agrega a la colección

HashSet:

➔ Esta implementación de Set almacena los elementos en una tabla hash.
➔ Esta implementación proporciona tiempos constantes en las operaciones básicas
siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash.
➔ Toda la lógica de verificación que el elemento no esté duplicado la maneja dentro del
mapa interno utilizado en la instancia del set → Las claves en un HashMap deben

ser únicas.
Implementación del método add(E e):
➔ public boolean add(E e) { return map.put(e, PRESENT)==null; }
➔ Si ese objeto ya está agregado devuelve el valor anterior de esa key pero si el objeto
no está y se agrega al HashMap devuelve null, entonces lo que hace el método add() es evaluar que devuelve y así se sabe si agrega o no el objeto

LinkedHashSet:

➔ Es similar a HashSet pero la tabla de dispersión es doblemente enlazada.
➔ Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo un coste constante, con la carga adicional que supone tener que gestionar los enlaces.Además tenemos enlaces entre los elementos y estos enlaces definen el orden en el que se insertaron en el conjunto.

TreeSet

➔ Esta implementación almacena los elementos ordenándolos en función de sus valores.
➔ Es bastante más lento que HashSet.
➔ Los elementos almacenados deben implementar la interfaz Comparable.Para poder
así tener el orden, una vez que ingresan a nuestro hassett le daría la ubicación a través de los valores
➔ Garantiza, siempre, un rendimiento de log(N) en las operaciones básicas,debido a la
estructura de árbol empleada para almacenar los elementos.

Genéricos:

➔ El término “genericidad” se refiere a una serie de técnicas que permiten escribir algoritmos o definir contenedores de forma que puedan aplicarse un amplio rango de
tipos de datos.
➔ Permite definir una clase o un método sin especificar el tipo de dato o parámetros,
de esta forma se puede cambiar la clase para adaptarla a diferentes usos sin tener que reescribirla.
➔ La programación genérica significa escribir un código que puedan reutilizar muchos
tipos diferentes de objetos

Declaración de tipos genéricos

➔ Una declaración de tipos genéricos puede tener múltiples tipos parametrizados, separados por comas.
➔ Todas las invocaciones de clases genéricas son expresiones de una clase. Al instanciar una clase genérica no se crea una nueva clase.
➔ No se puede usar un tipo de datos genérico en la creación de objetos y arreglos.

➔ Dentro de una definición de clases, el tipo genérico puede aparecer en cualquier declaración no estática donde se podría utilizar cualquier tipo de datos concreto.

Declaración de una clase genérica

➔ Una clase genérica o parametrizable es una clase con una o más variables de tipo
genérico.
public class MiClase { ... }
➔ Donde “MiClase” es el nombre de la clase genérica y “tipo_genérico” es el tipo parametrizado genérico

Enum:

➔ Constantes definidas de esta manera hacen el código más legible, permiten verificación en tiempo de compilación, documentan por adelantado la lista de valores
aceptados y evitan el comportamiento inesperado si es que se reciben valores no válidos.
➔ Un tipo enumerado restringe los posibles valores que puede tomar una variable. Esto ayuda a reducir los errores en el código y permite algunos usos especiales interesantes.
➔ Por convención, los nombres de los valores que puede tomar se escriben en letras
mayúsculas
➔ Un tipo enumerado puede ser declarado dentro o fuera de una clase, pero no dentro
de un método. Por tanto no podemos declarar un enum dentro de un método main;
➔ Declararemos un tipo enumerado como una clase aparte o dentro de otra, pero fuera
de cualquier método o constructor
➔ Se dispone automáticamente de métodos especiales como por ejemplo el método
values(), que el compilador agrega automáticamente cuando se crea un enum. Este método devuelve un array conteniendo todos los valores del enumerado en el orden en que son declarados
Operador “==”:
➔ Usando el tipo enum nos aseguramos que sólo exista una instancia de la constante
en la JVM. por lo tanto se puede usar el operador “==” para comparar dos variables del mismo tipo enum.

Excepciones:

➔ Una excepción en Java es un error o situación “excepcional” que se produce

durante
la ejecución de un programa.
➔ Ejemplos:
- Leer un archivo que no existe
- Acceder al valor N de una colección que tiene menos de N elementos.
- Enviar o recibir información por red mientras se produce un error de conectividad
➔ Todas las excepciones en Java se representan a través de objetos que heredan de
java.lang.Throwable.
➔ Las excepciones proporcionan una forma clara de comprobar posibles errores sin
oscurecer el código.
➔ Convierten las condiciones de error que un método puede señalar en una parte explícita del contrato del método.

Tipos de excepciones:

➔ Las excepciones también son objetos.
➔ Todos los tipos de excepción deben extender de la clase Throwable o de alguna de
sus subclases.
➔ De Throwable nacen dos ramas: Error y Exception.
1) Error representa errores de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos. Por ejemplo: errores de la JVM, desbordamiento de buffer.
2) Exception representa aquellos errores que normalmente solemos gestionar.
➔ Por convenio, los nuevos tipos de excepción extienden a Exception.
➔ De Exception nacen múltiples ramas: ClassNotFoundException, IOException, ParseException, SQLException → Excepciones checked.
➔ RuntimeException es la única excepción unchecked.

Excepciones checked (comprobadas):

➔ Una excepción de tipo checked representa un error del cual técnicamente podemos
recuperarnos.
➔ Son todas las situaciones que son totalmente ajenas al propio código, por ejemplo:
fallo de una operación de lectura/escritura.
➔ Este tipo de excepciones deben ser capturadas o relanzadas.

Excepciones unchecked (no comprobadas):

➔ Representan errores de programación. Por ejemplo: intentar leer de un array de N
elementos un elemento que se encuentra en una posición mayor que N.

Lanzamiento de excepciones:

➔ Las excepciones se lanzan utilizando la palabra reservada throw: throw AException
➔ AException debe ser un objeto Throwable.
➔ Si se lanza la excepción no se regresa al flujo normal del programa.
➔ Las excepciones son objetos, por lo tanto se debe crear una instancia antes de lanzarse.

Transferencia de control:

➔ Una vez que se produce una excepción, las acciones que hubiera detrás del punto
donde se produjo la excepción no tienen lugar.
➔ Las acciones que sí se ejecutarán serán las contenidas dentro de los bloques catch
y finally.
➔ Las sentencias dentro de la cláusula try se ejecutan hasta que se lance una excepción o hasta que finalice con éxito.
➔ Si se lanza una excepción, se examinan sucesivamente las sentencias de la cláusula catch.
➔ La cláusula finally de una sentencia proporciona un mecanismo para ejecutar una
sección de código, se lance o no una excepción.
➔ Generalmente la cláusula finally se utiliza para limpiar el estado interno o para liberar
recursos, como archivos abiertos o cerrar conexiones a bases de datos

Cláusula throws:

➔ Se utiliza para declarar las excepciones checked que puede lanzar un método.
➔ Se pueden declarar varias, separadas por comas.
➔ RuntimeException y Error son las únicas excepciones que no hace falta incluir en las
cláusulas throws.
● Si se invoca a un método que tiene una excepción checked en su cláusula throws, existen tres opciones:
1) Capturar la excepción y gestionarla
2) Capturar la excepción y transformarla en una de nuestras excepciones.
3) Declarar la excepción en la cláusula throws y hacer que otro método la gestione.
● No se permite que los métodos redefinidos declaren más excepciones checked en la
cláusula throws que las que declara el método.
● Se pueden lanzar subtipos de las excepciones declaradas ya que podrán ser capturadas en el bloque catch correspondiente a su supertipo.
● Si una declaración de un método se hereda en forma múltiple, la cláusula throws de
ese método

Malas Prácticas:

➔ No hacer en un try la lectura de un fichero y esperar a que falle, sino tiene que haber
un método separado para el manejo del archivo más allá de que estos métodos utilicen las excepciones y después las lancen o las cacheen propiamente.
➔ No crear métodos que arrojen excepción en general

Buenas prácticas:

➔ Utilizar excepciones que ya existen.
➔ Lanzar excepciones de acuerdo al nivel de abstracción en el que nos encontramos.
➔ Documentar con Javadoc todas las excepciones que se lanzan en los métodos

Archivo:

➔ Se puede definir a un archivo como un conjunto de bits almacenados en un dispositivo, y accesible a través de un camino de acceso (pathname) que lo identifica.
➔ Los archivos suelen organizarse en estructuras jerárquicas de directorios. Estos directorios son contenedores de ficheros (y de otros directorios) que permiten organizar los datos del disco.
➔ Los archivos suelen tener una serie de metadatos asociados, como pueden ser su
fecha de creación, la fecha de última modificación, etc.
➔ El parámetro “path” indica el camino hacia el directorio donde se encuentra el archivo, y “name” indica el nombre del archivo.
➔ Si el archivo existe, es decir, si la función exists() devuelve true, entonces podemos
obtener información acerca del archivo.
➔ Tenemos, además, los correspondientes métodos setReadable(), setReadOnly(), setWritable() y setExcutable() que nos permiten cambiar las propiedades del archivo siempre que seamos los propietarios del mismo o tengamos permisos para hacerlo.
➔ La siguiente comprobación útil que se puede hacer con un File es determinar si es
un archivo normal o es un directorio. Si es un directorio, podemos obtener un listado de los ficheros contenidos en él.
¿Por qué usar archivos?
➔ Con frecuencia tendremos que guardar los datos de nuestro programa para poder
recuperarlos más adelante. Los archivos se usan cuando el volumen de datos no es relativamente muy elevado.

Concepto de Buffering:

➔ “Si no hubiera buffers, sería como comprar sin un carrito: debería llevar los productos uno a uno hasta la caja. Los buffers te dan un lugar en el que dejar temporalmente las cosas hasta que esté lleno. Por eso se realizarán menos viajes cuando usas el carrito”
➔ Cualquier operación que implique acceder a memoria externa es muy costosa.
➔ Un buffer es una estructura de datos que permite el acceso por trozos a una

colección de datos.
➔ Los buffers son útiles para evitar almacenar en memoria grandes cantidades de datos, en lugar de ello, se va pidiendo al buffer porciones pequeñas de los datos que
se van procesando por separado. También resultan muy útiles para que la aplicación pueda ignorar los detalles concretos de eficiencia de hardware subyacente, la aplicación puede escribir al buffer cuando quiera, que ya se encargará el buffer de escribir a disco siguiendo los ritmos más adecuados y eficientes.

Escribir archivo:

➔ Para abrir un archivo usaremos un BufferedWriter, que se apoya en un FileWriter,
que a su vez usa un "File" al que se le indica el nombre del archivo.
➔ En el caso de un archivo de texto, no escribiremos con "println()", como hacíamos en
pantalla, sino con "write()". Cuando queramos avanzar a la línea siguiente, deberemos usar "newLine()".

Leer archivo:

➔ Para leer de un fichero de texto usaremos "readLine()", que nos devuelve un String.
Si ese String es null, quiere decir que se ha acabado el archivo y no se ha podido leer nada. Por eso, lo habitual es usar un "while" para leer todo el contenido de un fichero.
➔ Existe otra diferencia con la escritura: no usaremos un BufferedWriter, sino un BufferedReader, que se apoyará en un FileReader.
➔ La clase java.io.BufferedReader resulta ideal para leer archivos de texto y procesarlos. Permite leer eficientemente caracteres aislados, arrays o líneas completas como Strings. Cada lectura a un BufferedReader provoca una lectura en el archivo correspondiente al que está asociado. Es el propio BufferedReader el que se va encargando de recordar la última posición del fichero leído, de forma que posteriores lecturas van accediendo a posiciones consecutivas del fichero.

Cerrar un Archivo:

➔ fEntrada.close();

Serialización:

➔ La serialización permite convertir cualquier objeto (que implemente la interfaz Serializable) en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original.
➔ La serialización es una característica añadida al lenguaje Java para dar soporte a:
La invocación remota de objetos , La persistencia.
➔ La invocación remota de objetos permite a los objetos que “viven” en otras máquinas

comportarse como si vivieran en nuestra propia máquina.
➔ Para la persistencia, la serialización nos permite guardar el estado de un componente en disco, abandonar el IDE y restaurar el estado de dicho componente cuando se vuelve a correr el IDE.

Interfaz Serializable

➔ Un objeto se puede serializar si implementa la interfaz Serializable. Esta interfaz no declara ninguna función, se trata de una interfaz vacía.
➔ Para hacer una clase serializable simplemente debemos implementar esta interfaz.
➔ Si dentro de la clase hay atributos que son de otras clases, éstos a su vez también
deben implementar Serializable.
➔ La serialización de un objeto como atributo de otro objeto se puede tomar como un
árbol, donde las hojas son objetos que forman parte de otro objeto como un atributo, y todos ellos son marcados como serializables, para así entonces serializar primero las hojas del árbol y finalizar con la raíz.

Modificador transient:

➔ Habrá ocasiones donde no será necesario incluir un atributo del objeto en la serialización, y para esto se encuentra el modificador transient.
➔ Este modificador le indica a la JVM que dicho atributo deberá ser exento de la serialización, en otras palabras, ignorará este atributo.

➔ Por otro lado, los atributos que lleven el modificador static nunca se tomarán en cuenta al serializar un objeto, ya que este atributo pertenece a la clase y no al objeto.

Serialización y archivos

Un uso de la serialización es la lectura y escritura de objetos en un
archivo.

ObjectOutputStream:

➔ Se utiliza para escribir objetos en un OutputStream en lugar de escribir el objeto convertido a bytes. De esta manera se encapsula el OutputStream en un ObjectOutputStream.
➔ Sólo los objetos que implementen la interfaz Serializable pueden escribirse en los streams.

ObjectInputStream:

➔ Se utiliza en conjunto con ObjectOutputStream para leer los objetos que fueron escritos.
➔ Se debe notar que al leer los objetos tal vez sea necesario realizar un casting sobre
los mismos.

JSON:

➔ JSON (JavaScript Object Notation) es un formato para intercambiar datos liviano,
basado en texto e independiente del lenguaje de programación fácil de leer tanto para seres humanos como para las máquinas.

➔ Puede representar dos tipos estructurados: objetos y matrices.
➔ Un objeto es una colección no ordenada de cero o más pares de nombres/valores.
➔ Una matriz es una secuencia ordenada de cero o más valores.
➔ Los valores pueden ser cadenas, números, booleanos, nulos y estos dos tipos estructurados.

Librerías para procesamiento de JSON:

➔ Jackson: es una librería de utilidad de Java que nos simplifica el trabajo de serializar
(convertir un objeto Java en una cadena de texto con su representación JSON), y deserializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON. Para ello usa básicamente la introspección de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la deserialización buscará un método “setName(String s)”.

➔ **Gson:**

el uso de esta librería se basa en el uso de una instancia de la clase Gson.
Dicha instancia se puede crear de manera directa (new Gson()) para transformaciones sencillas o de forma más compleja con GsonBuilder para añadir distintos comportamientos.