

# Stream (Java Platform SE 8 )

List

# Maps, Set y List

Los **arraylist s**on dinámicos y cada valor dentro tiene una referencia en un índice para recorrerlos en un for o en un for each.

public class ListExample {

public static void main(String[] args) {

// Crear una List de enteros

List<Integer> numeros = new ArrayList<>();

// Agregar elementos a la List numeros.add(10);

// Acceder a elementos por índice int primerElemento = numeros.get(0);

// Recorrer la List

for (int numero : numeros) { System.out.println(numero);

}

// Eliminar un elemento numeros.remove(1);

}

}

## Sets:

HASHSET

no guarda duplicados y no tienen orden de almacenamiento.

TREESET

Se almacenan de MAYOR A MENOR y no tienen duplicados tampoco

LINKEDHASHSET

No hay duplicados y estan en orden de agregacion, como en una lista.

import java.util.HashSet; import java.util.Set; public class SetExample {

public static void main(String[] args) {

// Crear un Set de cadenas Set<String> nombres = new HashSet<>();

// Agregar elementos al Set nombres.add("Juan"); nombres.add("María"); nombres.add("Pedro");

nombres.add("María"); // No se permite duplicados

// Recorrer el Set

for (String nombre : nombres) { System.out.println(nombre);

}

// Verificar si un elemento existe en el Set boolean contienePedro = nombres.contains("Pedro");

System.out.println("Contiene Pedro: " + contienePedro);

// Eliminar un elemento nombres.remove("Juan");

System.out.println("Después de eliminar a Juan: " + nombres);

}

}

## Map

Un Map es una estructura de datos que asocia claves (keys) con valores (values). Cada clave es única y se utiliza para acceder a su correspondiente valor. En Java, la interfaz más común para los Maps es java.util.Map. Algunas implementaciones conocidas son HashMap y TreeMap. import java.util.HashMap;

import java.util.Map; public class MapExample {

public static void main(String[] args) {

// Crear un Map de cadenas a enteros Map<String, Integer> notas = new HashMap<>();

// Agregar elementos al Map notas.put("Juan", 90);

notas.put("María", 85);

notas.put("Pedro", 95);

// Acceder a un valor por clave

int notaDeJuan = notas.get("Juan"); System.out.println("Nota de Juan: " + notaDeJuan);

// Recorrer el Map

for (Map.Entry<String, Integer> entry : notas.entrySet()) { String nombre = entry.getKey();

int nota = entry.getValue(); System.out.println(nombre + ": " + nota);

}

// Verificar si una clave existe en el Map

boolean contieneMaria = notas.containsKey("María"); System.out.println("Contiene a María: " + contieneMaria);

// Eliminar una entrada notas.remove("Pedro");

System.out.println("Después de eliminar a Pedro: " + notas);

}

}

## Importante

La diferencia principal entre HashSet/HashMap y TreeSet/TreeMap radica en la forma en que se almacenan y ordenan los elementos.

HashSet / HashMap:

HashSet es una implementación de la interfaz Set que utiliza una tabla hash para almacenar los elementos. Esto significa que no hay un orden específico en los elementos almacenados.

HashMap es una implementación de la interfaz Map que también utiliza una tabla hash, pero almacena pares de clave-valor. Tampoco hay un orden específico en los pares clave-valor almacenados.

La ventaja principal de HashSet y HashMap es su eficiencia en términos de búsqueda, inserción y eliminación de elementos. Sin embargo, no garantizan ningún orden particular de los elementos almacenados.

TreeSet / TreeMap:

TreeSet es una implementación de la interfaz Set que mantiene los elementos ordenados en orden ascendente o según un comparador personalizado. Utiliza una estructura de árbol (generalmente un árbol rojo-negro) para almacenar y organizar los elementos.

TreeMap es una implementación de la interfaz Map que también mantiene los pares clave-valor ordenados según la clave.

La ventaja principal de TreeSet y TreeMap es que los elementos se almacenan en orden, lo que facilita la iteración en un orden específico. Sin embargo, esta ventaja conlleva una pequeña pérdida de eficiencia en comparación con HashSet y HashMap.

En resumen, mientras que HashSet/HashMap se enfocan en la eficiencia y no garantizan un orden específico, TreeSet/TreeMap se centran en mantener los elementos ordenados, lo que puede ser útil en situaciones donde el orden es importante. La elección entre ellos dependerá de tus necesidades

específicas en términos de eficiencia y ordenamiento.

# Excepciones

## Try-catch

En un metodo, puedes poner las posibles excepciones que te salte, que pueden ser creadas en una clase abstracta. Cuando creas una excepcion, simplemente le creas dos constructores, uno vacio y otro que reciba un parametro de entrada tipo String que se lo pasa al padre, que va a ser el mensaje de error por defecto. Luego en el metodo que implementa esa excepcion dices que en un caso excepcional tire de la excepcion con ese mensaje de error por defecto que enviabamos al padre. Cuando llamas al metodo desde otro sitio, tienes que hacerlo en un try catch porque en caso de saltar la excepcion pones el mensaje que quieres que diga si salta con un SOUT, tambien puedes llamar a metodos en el catch para que vuelva a introducir los parametros por los que le ha podido saltar el error o para mirar que excepcion especifica ha saltado por consola.

## Importante

El método main() es el punto de entrada de un programa y es responsabilidad del programador manejar las excepciones dentro de él utilizando bloques try-catch para controlar el flujo y

solucionar los errores en el códigoFicheros

# For each

El bucle "for-each" (también conocido como "enhanced for loop" en Java) es una forma conveniente de recorrer elementos en una colección o matriz (array) en Java. A diferencia de

otros tipos de bucles, como el "for" tradicional, el bucle "for-each" no utiliza un contador o un índice explícito para iterar sobre los elementos. En su lugar, se encarga automáticamente de recorrer cada elemento en la colección. Veamos cómo se utiliza el bucle "for-each" en Java. for (tipoDeElemento elemento : colección) {

// Cuerpo del bucle

// Se ejecuta para cada elemento en la colección

}

# Lambda

En programación, una expresión lambda es una forma concisa de representar una función anónima, es decir, una función que no tiene un nombre asociado. Las expresiones lambda son utilizadas para implementar el paradigma de programación funcional en Java, permitiendo

tratar las funciones como objetos de primera clase. Esto significa que puedes pasar una función como argumento, devolverla como resultado y almacenarla en una variable.

La sintaxis básica de una expresión lambda en Java es la siguiente:

(parametros) -> { cuerpo de la función }

Donde "parámetros" representa los parámetros de la función (si los hay) y "cuerpo de la

función" contiene las instrucciones que se ejecutarán al llamar a la función. A continuación, te mostraré algunos ejemplos para ilustrar cómo funcionan las expresiones lambda en diferentes contextos.

1. Expresiones lambda sin parámetros:

() -> {

// Código de la función

}

Este ejemplo representa una función sin parámetros. Puedes colocar el código que desees dentro del cuerpo de la función.

2. Expresiones lambda con un parámetro:

(parametro) -> {

// Código de la función que utiliza el parámetro

}

Aquí, "parámetro" representa el único parámetro de la función. Puedes acceder y utilizar este parámetro dentro del cuerpo de la función.

3. Expresiones lambda con múltiples parámetros:

(parametro1, parametro2) -> {

// Código de la función que utiliza los parámetros

}

En este caso, "parametro1" y "parametro2" son los parámetros de la función. Puedes utilizar ambos parámetros en el cuerpo de la función.

Una vez que hayas definido una expresión lambda, puedes utilizarla en diferentes contextos, como en métodos de orden superior, interfaces funcionales y colecciones.

Métodos de orden superior:

1. Los métodos de orden superior son aquellos que toman una función como argumento o devuelven una función. Puedes utilizar expresiones lambda para pasar una función como argumento a un método. Por ejemplo, supongamos que tienes un método filtrar que toma una lista y una función de filtro, y devuelve una nueva lista con los elementos que cumplen el criterio de filtro. Puedes utilizar una expresión lambda para definir el

filtro en el lugar donde llamas al método filtrar.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); List<Integer> numerosPares = filtrar(numeros, (numero) -> numero % 2

== 0);

En este ejemplo, (numero) -> numero % 2 == 0 es una expresión lambda que representa una función que toma un número y devuelve true si es par.

1. Interfaces funcionales:

Java proporciona interfaces funcionales predefinidas que se pueden utilizar como tipos para expresiones lambda. Una interfaz funcional es una interfaz que contiene exactamente un método abstracto. Puedes utilizar una expresión lambda para

implementar el método abstracto de una interfaz funcional. Por ejemplo, la interfaz funcional Runnable se utiliza para representar una tarea que se ejecutará en un hilo separado. Tiene un único método abstracto llamado run(). Puedes utilizar una expresión lambda para implementar este método.

Runnable runnable = () -> {

// Código de la tarea que se ejecutará en un hilo separado

};

Aquí, la expresión lambda () -> { // Código de la tarea } se utiliza para implementar el método run() de la interfaz Runnable.

Además de las interfaces funcionales predefinidas, también puedes crear tus propias

interfaces funcionales. Una interfaz funcional personalizada es aquella que defines con un único método abstracto y puedes utilizar expresiones lambda para implementar ese método en diferentes contextos.

1. Colecciones:

Las expresiones lambda también son útiles para operaciones en colecciones, como

filtrar, mapear, reducir, entre otras. Estas operaciones se pueden realizar utilizando el nuevo API de Streams introducido en Java 8. Por ejemplo, supongamos que tienes una lista de números y quieres filtrar solo los números pares y luego imprimirlos. Puedes usar una expresión lambda con el método filter() y forEach() de la clase Stream para

lograrlo.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); numeros.stream()

.filter(numero -> numero % 2 == 0)

.forEach(System.out::println);

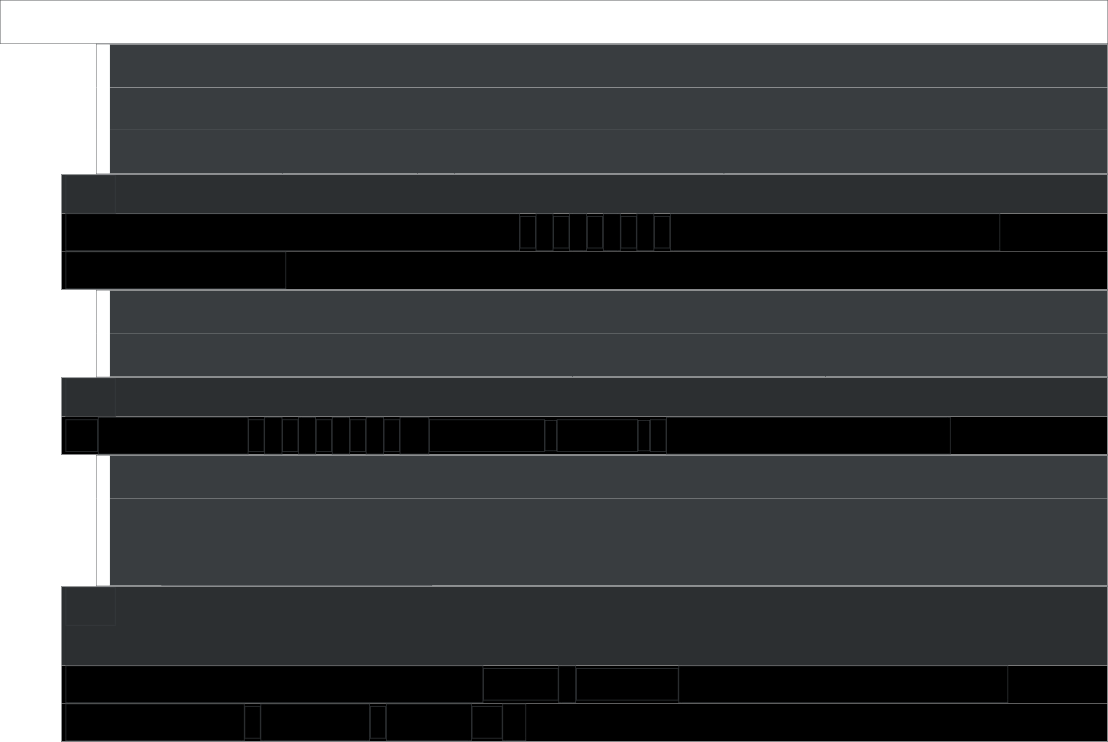
Aquí, la expresión lambda numero -> numero % 2 == 0 se utiliza como el predicado de filtro para seleccionar los números pares. Después de aplicar el filtro, se utiliza el método forEach() para imprimir cada número par.

# Stream



En Java, un Stream es una secuencia de elementos que se puede procesar de manera secuencial o paralela. Proporciona una forma declarativa y funcional de realizar operaciones en colecciones de datos, como listas o arreglos. Los Streams permiten escribir código más conciso y legible, evitando bucles explícitos y condicionales.

Aquí tienes los pasos básicos para trabajar con Streams en Java:



List<Integer> numeros = Arrays.asList( 1 , 2

* A partir de un arreglo: Puedes crear un Stream a partir de un arreglo utilizando el método **Arrays.stream()**. Por ejemplo:

Copy code

"Hola" ,

).limit( 10 );

1

, n -> n +

0

Stream.iterate(

); Stream<Integer> stream =

"Mundo"

Stream<String> stream = Stream.of(

java

* Utilizando métodos generadores: También puedes crear Streams utilizando métodos generadores, como **Stream.of()** o **Stream.iterate()**. Por ejemplo:

Arrays.stream(arreglo);

=

stream

IntStream

};

5

,

4

,

3

,

2

,

1

[] arreglo = {

int

Copy code

java

numeros.stream();

); Stream<Integer> stream =

5

,

4

,

3

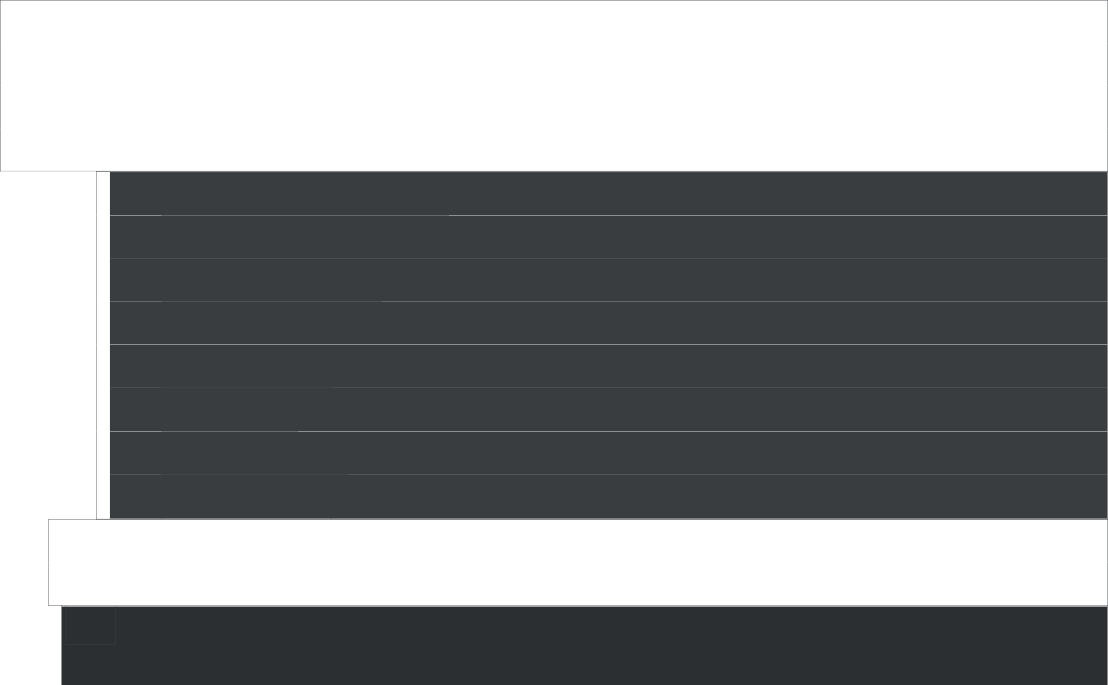
,

Copy code

java

* A partir de una colección: Puedes crear un Stream a partir de una colección existente, como una lista o un conjunto, utilizando el método **stream()** o **parallelStream()**. Por ejemplo:

1. Creación de un Stream:



Copy code

java

Por ejemplo, puedes filtrar los números pares y duplicarlos usando las operaciones intermedias:

* **filter(Predicate)**: Filtra los elementos del Stream según un predicado.
* **map(Function)**: Transforma cada elemento del Stream mediante una función.
* **distinct()**: Elimina los elementos duplicados en el Stream.
* **sorted()**: Ordena los elementos del Stream.
* **limit(long)**: Limita el número de elementos del Stream.
* **skip(long)**: Salta los primeros elementos del Stream.

2. Operaciones intermedias: Después de crear un Stream, puedes realizar operaciones intermedias en él. Estas operaciones no producen un resultado final, pero transforman o filtran los elementos del Stream. Algunas operaciones intermedias comunes son:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| List<Integer> numeros = Arrays.asList( | | | | | 1 | , | 2 | , | 3 | , | 4 | , | 5 | ); numeros.stream() .filter(n -> n % | 2 |
| == | 0 | ) .map(n -> n \* | 2 | ) .forEach(System.out::println); | | | | | | | | | |  | |

**MAP:**

Te da los valores de cualquier método que devuelva un atributo de un objeto que elijas.

Ejemplo:

public List<Persona> consulta(String ciudad){

//coges la lista de personas ya creada

personas.stream()

.filter(persona -> persona.getNombre().equals(ciudad))

.map(persona::getNombre)

.distinct()

.collect(Collections.tolist)

}

Este ejemplo te da el nombre de las personas que viven en determinadas ciudades sin udplicados.

**Operaciones terminales**: Después de aplicar las operaciones intermedias, debes realizar una operación terminal para obtener un resultado final. Algunas operaciones terminales comunes son básicamente para filtrar datos de una lista dentro de unos parametros que nosotros le indiquemos. Por ejemplo:

Dentro de esta lista de personas, para cada persona de la lista que

tenga un nombre igual al indicado se enseñe:

public List<Persona> consulta(String nombre){

//coges la lista de personas ya creada

personas.stream()

.filter(persona -> persona.getNombre().equals(nombre))

.collect(Collections.tolist)

}

Al final la metes en una colección que básicamente lo que hace es recabar todos los datos filtrados a una lista auxiliar que es la que puedes returnear directamente o igualarla a otra lista

5. Manejo de archivos

Para cargar un archivo en memoria RAM al inicio del programa, puedes utilizar la clase FileInputStream si estás trabajando con archivos binarios, o la clase FileReader si estás

trabajando con archivos de texto. Estas clases te permiten leer el contenido del archivo y almacenarlo en una estructura de datos en memoria RAM.

Para escribir en un archivo, puedes utilizar la clase FileOutputStream si estás trabajando con archivos binarios, o la clase FileWriter si estás trabajando con archivos de texto. Estas clases te permiten escribir datos en el archivo.

Para reescribir en un archivo sin sobreescribir colecciones, puedes utilizar la clase

RandomAccessFile. Esta clase te permite acceder a una posición específica en el archivo y sobrescribir datos sin afectar a otras partes del archivo. Puedes utilizar la clase

FileOutputStream o FileWriter para guardar los datos en un archivo. Para archivos no Binarios:

import java.io.\*; public class Archivo {

public static void main(String[] args) { try {

// Abrir el archivo para leer

File archivo = new File("archivo.txt"); FileReader lector = new FileReader(archivo);

// Leer el contenido del archivo int caracter;

while ((caracter = lector.read()) != -1) { System.out.print((char) caracter);

}

// Cerrar el archivo de lectura lector.close();

// Abrir el archivo para escribir

FileWriter escritor = new FileWriter(archivo, true);

// Escribir en el archivo escritor.write("Hola, mundo!");

// Cerrar el archivo de escritura escritor.close();

} catch (IOException e) {

System.out.println("Ocurrió un error al manipular el archivo."); e.printStackTrace();

}

}

}

//Otra Implementacion:

public static void escribirFichero(List<Alojamiento> cesas) { File fichero = new File(ficheronob);

try (BufferedWriter bufferedwriter = new Bufferedwriter(new FileWriter(fichero))) {

for (Alojamiento casa : casas) { bufferedWriter.write(casa.toStringFichero()); bufferedWriter.newLine();

}

} catch (IOException e) {

throw new RuntimeException(e);

}

}

Para archivos Binarios:

import java.io.\*;

public class ArchivoObjeto {

public static void main(String[] args) {

// Crear un objeto para escribir en el archivo MiObjeto objeto = new MiObjeto("Hola, mundo!"); try {

// Abrir el archivo para escribir

FileOutputStream archivoSalida = new FileOutputStream("archivo.obj"); ObjectOutputStream escritorObjeto = new

ObjectOutputStream(archivoSalida);

// Escribir el objeto en el archivo escritorObjeto.writeObject(objeto);

// Cerrar el archivo de escritura escritorObjeto.close();

// Abrir el archivo para leer

FileInputStream archivoEntrada = new FileInputStream("archivo.obj"); ObjectInputStream lectorObjeto = new ObjectInputStream(archivoEntrada);

// Leer el objeto del archivo

MiObjeto objetoLeido = (MiObjeto) lectorObjeto.readObject();

// Cerrar el archivo de lectura lectorObjeto.close();

// Acceder a los datos del objeto leído System.out.println(objetoLeido.getMensaje());

} catch (IOException e) {

System.out.println("Ocurrió un error al manipular el archivo."); e.printStackTrace();

} catch (ClassNotFoundException e) { System.out.println("Error al leer el objeto del archivo."); e.printStackTrace();

}

}

}

// Clase de ejemplo para el objeto a ser escrito y leído class MiObjeto implements Serializable {

private String mensaje;

public MiObjeto(String mensaje) { this.mensaje = mensaje;

}

public String getMensaje() { return mensaje;}}

# Estructura de capas Funcional

LOMBOCK

Pones en las clases objeto @AllArgsConstructor y @Data para no tener que escribirlo todo.

Puedes pone ren una clase @log como en costantes por ejemplo y usas log.warning(mensaje).

*public* List<Vivienda> viviendasPorCalleNumero(String provincia) {

        return database.getListaViviendas().stream()

                .filter(Vivienda -> Vivienda.getProvincia()

.equals(provincia))

                .sorted(new ComparacionPorCalleMetros())

                .collect(Collectors.toList());

    }

*public* boolean actualizarm2(int id, double m2) {

        Vivienda h = database.getListaViviendas().stream()

                .filter(vivienda -> vivienda.getId()==id)

.findFirst().orElse(null);

        if (h != null) {

            try {

                Comprobacion.m2Ok(m2);

                h.setM2(m2);

                return true;

            } catch (m2Exception e) {

                throw new RuntimeException(e);

            }

        }

        return false;

    }

*public* List<Vivienda> listadoOrdenadoViviendasCalle(String calle,

boolean ascendente) {

        List<Vivienda> viviendas = database.getListaViviendas()

.stream()

.filter(v->v.getCalle().equalsIgnoreCase(calle))

.sorted(new ComparacionPorCalleMetros()).toList();

        if (!ascendente) {

            viviendas = new ArrayList<>(viviendas);

            Collections.reverse(viviendas);

        }

        return viviendas;

    }

    @Override

*public* List<Vivienda> getListaviviendasProvincia(String provincia) {

        return database.getListaViviendas().stream()

.filter(v->v.getProvincia().equalsIgnoreCase(provincia))

.toList();

    }

*public* List<Vivienda> getListaViviendasProvincia(String provincia) {

        return database.getListaViviendas().stream()

                .filter(Vivienda -> Vivienda.getProvincia()

.equals(provincia))

                .collect(Collectors.toList());

    }

*public* void removeVivienda(Vivienda vivienda) {

        database.getListaViviendas().remove(vivienda);

    }

*public* void setViviendas(List<Vivienda> viviendas) {

        database.setListaViviendas(viviendas);

    }

*public* boolean isEmptyViviendasList() {

        return database.getListaViviendas().isEmpty();

    }

*public* void escribirArrayListBinario(ArrayList<Vivienda> listaObjetos) {

        try {

            FileOutputStream archivoBinario =

new FileOutputStream("ruta\_del\_archivo");

ObjectOutputStream escritura =

new ObjectOutputStream(archivoBinario);

            escritura.writeObject(listaObjetos);

            escritura.close();

            archivoBinario.close();

            System.out.println

("ArrayList guardado en el archivo binario correctamente.");

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

*public* *class* ComparacionPorCalleMetros *implements* Comparator<Vivienda> {

    @Override

*public* int compare(Vivienda arg0, Vivienda arg1) {

        int aux=arg0.getCalle().compareTo(arg1.getCalle());

        if(aux==0){

            aux=Double.compare(arg0.getM2(),arg1.getM2());

        }

        return aux;

    }

}

*public* *class* m2Exception *extends* Exception {

*public* m2Exception() {

        super("Los metros cuadrados deben ser un número mayor que 0");

    }

}

*public* *class* Comprobacion {

*public* *static* void m2Ok(double m2) *throws* m2Exception {

        if (m2 < 1) {

            throw new m2Exception();

        }

    }

}

*public* void cargarUsuarios(String file) {

        Scanner scanner = null;

        comunidad = new ArrayList<>();

        try {

            scanner = new Scanner(new File(file));

            while (scanner.hasNextLine()) {

                User usuario = new User(scanner.nextLine());

                comunidad.add(usuario);

            }

        } catch (FileNotFoundException e) {

            System.out.println("Archivo no encontrado");

        }

    }*// ARCHIVOS NO BINARIOS, READ*

public void guardarEnArchivo(String rutaArchivo) {

try (PrintWriter writer = new PrintWriter(new FileWriter(rutaArchivo))) {

writer.println(this.toString());

System.out.println("El usuario se ha guardado correctamente en el archivo.");

} catch (IOException e) {

System.err.println("Error al guardar el usuario en el archivo: " + e.getMessage());

}

}