Stream (Java Platform SE 8)

1. Maps, Set y List

List

Una List es una colección ordenada de elementos en la que se permite la duplicación de valores. Puedes acceder a los elementos de una List por su índice, lo que significa que los elementos están ordenados según el orden en el que se agregaron. En Java, la interfaz más común para las Listas es java.util.List. Algunas implementaciones conocidas de la interfaz List son ArrayList y LinkedList:

```
import java.util.ArrayList;
import java.util.List;
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);
        numeros.add(20);
        numeros.add(30);
        // Acceder a elementos por índice
        int primerElemento = numeros.get(0);
        System.out.println("Primer elemento: " + primerElemento);
        // Recorrer la List
        for (int numero : numeros) {
            System.out.println(numero);
        }
        // Eliminar un elemento
        numeros.remove(1);
        System.out.println("Después de eliminar un elemento: " + numeros);
    }
}
```

Sets:

Un Set es una colección que no permite elementos duplicados. No hay un orden definido en un Set, lo que significa que no puedes acceder a los elementos por su posición. En Java, la interfaz más común para los Sets es java.util.Set. Algunas implementaciones conocidas son HashSet y TreeSet.

```
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.

2. Excepciones

Try-catch

El bloque try-catch se utiliza para controlar excepciones en Java y garantizar un manejo adecuado de los errores que puedan ocurrir durante la ejecución de un programa. Permite detectar excepciones específicas y proporciona una forma de manejarlas de manera controlada.

```
try {
    // Código que puede generar una excepción
} catch (TipoDeExcepcion1 excepcion1) {
    // Manejo de excepción 1
} catch (TipoDeExcepcion2 excepcion2) {
    // Manejo de excepción 2
} finally {
    // Bloque opcional para ejecutar código siempre,
    // independientemente de si se produjo una excepción o no
}
```

Aquí hay una explicación paso a paso del funcionamiento del bloque try-catch:
El código que puede generar una excepción se coloca dentro del bloque try.

1. Si se produce una excepción en el bloque try, la ejecución se detiene inmediatamente y se busca un bloque catch correspondiente.
2. Cada bloque catch tiene un tipo de excepción asociado, y si la excepción generada coincide con el tipo especificado, se ejecuta el código dentro de ese bloque catch. Puedes tener múltiples bloques catch para manejar diferentes tipos de excepciones.
3. Una vez que se ha manejado una excepción en un bloque catch, el programa continúa su ejecución normal después del bloque try-catch. Si no se encuentra un bloque catch correspondiente para la excepción generada, el programa se detendrá y mostrará un mensaje de error.

El bloque finally es opcional y se utiliza para ejecutar código que siempre debe ejecutarse, independientemente de si se produjo una excepción o no. Se coloca después de los bloques catch.

Crear Excepciones

Ahora, para crear tus propias excepciones personalizadas, puedes seguir estos pasos:

1. Crear una clase que extienda de la clase Exception o alguna de sus subclases, como RuntimeException. Esto se hace para definir tu propia clase de excepción personalizada.
   ```
   public class MiExcepcion extends Exception {
   // Puedes agregar constructores y métodos adicionales según tus necesidades
   }
   ```
2. Dentro de tu código, cuando detectes una condición excepcional, puedes lanzar tu propia excepción personalizada usando la palabra clave throw. Esto debe hacerse dentro del bloque try o dentro de un método que indique que lanza esa excepción.
   ```
   if (condicionExcepcional) {
   throw new MiExcepcion("Este es un mensaje de error opcional");
   }
   ```
3. Para manejar tu excepción personalizada, puedes usar un bloque catch específico para esa excepción o uno más general si deseas manejar diferentes tipos de excepciones juntas.
   ```
   try {
   // Código que puede lanzar MiExcepcion
   } catch (MiExcepcion e) {
   // Manejo de tu excepción personalizada
   } catch (OtraExcepcion e) {
   // Manejo de otra excepción
   } catch (Exception e) {
   // Manejo de excepciones generales
   }
   ```
   En el ejemplo anterior, si se lanza una instancia de MiExcepcion, el programa capturará esa excepción en el primer bloque catch y ejecutará el código correspondiente para manejarla. Si se lanza una instancia de OtraExcepcion, se capturará en el segundo bloque catch. Si ninguna de las excepciones anteriores coincide, se capturará en el último bloque catch, que maneja excepciones generales.
   Recuerda que el orden de los bloques catch es importante. Debes capturar las excepciones más específicas primero y las más generales al final.

Throws

El uso de la palabra clave throws en Java se utiliza para indicar que un método puede lanzar una o más excepciones verificadas. Las excepciones verificadas son aquellas que deben ser declaradas en la firma del método o manejadas dentro del método utilizando un bloque try-catch.

Cuando un método utiliza throws para declarar una excepción, significa que el método no manejará la excepción en sí, sino que la propagará a quien invoque ese método para que sea manejada en un nivel superior.

Aquí tienes un ejemplo de cómo se utiliza la palabra clave throws en la firma de un método:

```
public void miMetodo() throws MiExcepcion {
    // Código que puede lanzar MiExcepcion
}
```

En este ejemplo, el método miMetodo() declara que puede lanzar una excepción de tipo MiExcepcion. Si se produce esa excepción dentro del método, no se manejará allí mismo, sino que se propagará hacia el código que invocó este método. Será responsabilidad del código que llama a miMetodo() manejar la excepción utilizando un bloque try-catch o propagarla hacia arriba utilizando también throws.
Aquí tienes un ejemplo de cómo se puede manejar una excepción lanzada por un método que utiliza throws:

```
public static void main(String[] args) {
    try {
        miMetodo();
    } catch (MiExcepcion e) {
        // Manejo de la excepción MiExcepcion
    }
}
```

En este caso, el método main() está llamando al método miMetodo(), que declara que puede lanzar una excepción MiExcepcion. Para manejar esa excepción, se utiliza un bloque try-catch en el método main().
Recuerda que solo las excepciones verificadas (aquellas que son subclases de Exception pero no de RuntimeException) requieren ser declaradas en la firma del método o manejadas dentro del método. Las excepciones no verificadas, como las subclases de RuntimeException, no requieren el uso de throws o try-catch.

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

3. 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
}
```

Aquí tienes una explicación paso a paso de cómo funciona:

- tipoDeElemento: Esto representa el tipo de datos de cada elemento en la colección. Por ejemplo, si estás recorriendo una matriz de enteros, tipoDeElemento sería int. Si estás recorriendo una lista de cadenas, tipoDeElemento sería String.
- elemento: Esto es una variable que se utilizará para representar cada elemento en la colección durante cada iteración del bucle. Puedes darle cualquier nombre válido que desees.
- colección: Esto representa la colección o matriz sobre la cual deseas iterar.

Durante cada iteración del bucle, la variable elemento tomará el valor del siguiente elemento en la colección. El bucle se repetirá automáticamente hasta que se hayan recorrido todos los elementos en la colección.

Aquí tienes un ejemplo práctico que muestra cómo se utiliza el bucle "for-each" para recorrer una matriz de enteros:

```
int[] numeros = {1, 2, 3, 4, 5};
for (int numero : numeros) {
    System.out.println(numero);
}
```

En este ejemplo, la variable numero toma el valor de cada elemento en la matriz numeros durante cada iteración del bucle. El bucle imprime cada número en una línea separada.
El bucle "for-each" también se puede utilizar con otras colecciones en Java, como listas, conjuntos o mapas. Aquí tienes un ejemplo de cómo recorrer una lista de cadenas:

```
List<String> palabras = new ArrayList<>();
palabras.add("Hola");
palabras.add("Mundo");
for (String palabra : palabras) {
    System.out.println(palabra);
}
```

En este caso, la variable palabra tomará el valor de cada elemento en la lista palabras. El bucle imprime cada palabra en una línea separada.
Recuerda que el bucle "for-each" solo permite iterar sobre los elementos de la colección en orden, de principio a fin. No proporciona acceso a un índice o posición específica en la colección.

4. 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.
2. 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.
3. 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.

4. 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:

1. Creación de un Stream:
   - 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:

```
javaCopy code
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5); Stream<Integer> stream = numeros.stream();
```

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

```
javaCopy code
int [] arreglo = {1, 2, 3, 4, 5}; IntStream stream = Arrays.stream(arreglo);
```

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

```
javaCopy code

Stream<String> stream = Stream.of("Hola", "Mundo"); Stream<Integer> stream = Stream.iterate(0, n -> n + 1).limit(10);
```

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:
   - `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.

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

```
javaCopy code
```

```
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);
```

3. Operaciones terminales: Después de aplicar las operaciones intermedias, debes realizar una operación terminal para obtener un resultado final. Algunas operaciones terminales comunes son:
    - **`forEach(Consumer)`**: Ejecuta una acción en cada elemento del Stream. Por ejemplo:

javaCopy code
```
List<String> nombres = Arrays.asList("Juan", "Pedro", "Ana");
nombres.stream().forEach(System.out::println);
```

    - **`count()`**: Cuenta el número de elementos del Stream. Por ejemplo:

javaCopy code
```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5); long cantidad = numeros.stream().count();
```

    - **`anyMatch(Predicate)`**: Comprueba si algún elemento cumple con un predicado. Por ejemplo:

javaCopy code
```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5); boolean hayPar = numeros.stream().anyMatch(n -> n % 2 == 0);
```

    - **`allMatch(Predicate)`**: Comprueba si todos los elementos cumplen con un predicado. Por ejemplo:

javaCopy code
```
List<Integer> numeros = Arrays.asList(2, 4, 6, 8, 10); boolean sonPares = numeros.stream().allMatch(n -> n % 2 == 0);
```

    - **`noneMatch(Predicate)`**: Comprueba si ningún elemento cumple con un predicado. Por ejemplo:

javaCopy code
```
List<Integer> numeros = Arrays.asList(1, 3, 5, 7, 9); boolean sonPares = numeros.stream().noneMatch(n -> n % 2 == 0);
```

    - **`collect(Collector)`**: Recolecta los elementos del Stream en una colección o valor. Por ejemplo, puedes recolectar los números pares en una lista:

javaCopy code
```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5); List<Integer> pares = numeros.stream().filter(n -> n % 2 == 0).collect(Collectors.toList());
```

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;}}
```

6. Estructura de capas Funcional