Contents

Modulo 4: Programacion avanzada en Java

Intro

En este punto ya estamos medianamente familiarizados con Java, tenemos nocion de como crear datos, variables, clases, metodos y tenemos un mayor conocimiento de como usar estructuras de control y ciclos (loops) para iterar arreglos.

Si crees que aun te hace falta un poco de practica, recuerda que puedes preguntar y compartir codigo en los grupos de telegram y discord.

Adicionalmente te comparto algunos ejercicios que puedes usar para practicar cada tema (algunos links podrian requerir una cuenta, pero eso te podria permitir probar tus soluciones en linea):

- Modulo 1: HackerRank Java Stdin and Stdout
- Modulo 2:
  - Variables y tipos de dato: HackerRank Stdin and Stdout 2
  - Estructuras de control:
    - HackerRank if-else
    - Switch problem (resuelvelo usando switch): PROBLEMA_SWITCH.md
    - Loops:
      - For Loop: HackerRank Loops 2
      - While Loop: HackerRank End-of-file
      - Metodos: Exercism: Lasagna
      - Arrays: HackerRank subarray
- Modulo 3: HackerRank Inheritance 2 En estos links puedes encontrar mas ejercicios y puedes navegar ejercicios mas faciles y dificiles. Recuerda que la práctica regular te lleva a la maestría. Cada repetición cuenta, así que sigue practicando y verás los resultados.

Programacion avanzada en Java

Un parentesis en la palabra package

Conforme nuestro codigo crezca nos veremos en la necesidad de separar nuestro proyecto en multiples clases o archivos para poder tener mayor control sobre de el.

Para poder tener un mayor control y no terminar con 200 archivos en una sola carpeta, facilitar la distribucion de nuestro proyecto final y darle identidad unica a nuestro proyecto, podemos usar `package`.

Hay algunas reglas a seguir para poder usar `package`

- `package` debe ser la primer linea de codigo (sin contar comentarios) en tu archivo
- La estructura de `package` esta delimitada por `.`: `package palabra.otrapalabra.nombre.grupo.etc`
- La linea debe terminar con `;`
- Si tu package es: `package hola.adios` tu archivo debe estar en la carpeta/directorio: `src/java/hola/adios`
- Idealmente, por buena practica, el nombre del package debe estar formado por letras minusculas
- Tu package, idealmente, debe incluir unicamente letras, numeros, puntos y guiones bajos (`_`)
- Ninguna parte del package debe iniciar con numero, en caso de querer incluir numeros al principio de algun bloque, se debe usar `_`
- Comunmente, si tienes un sitio web, tu `package` debe ser tu *dominio* en orden inverso: si tu sitio web es `codigofacilito.com`, tu `package` sera `package com.codigofacilito`

Puedes consultar estas reglas y la documentacion completa en la documentacion de Oracle

El poder del JDK

El JDK viene cargado de clases que pueden hacer mucho del trabajo que necesitamos.

Un ejemplo rapido de ello es la clase `Scanner`. La clase Scanner es una clase que nos permite "*leer*" datos de alguna fuente, dicha fuente puede ser un archivo, la conosola/terminal o algun otro productor de datos. Una gran ventaja es que la clase `Scanner` sabe como separar elementos identificandolos por un "delimitador", esto es, la clase puede identificar si los elementos estan separados por un espacio o, podemos, decirle si los datos estan separados por una coma (`,`) o cualquier otro simbolo.

Aqui esta la documentacion oficial de la la clase Scanner

Podriamos hacer un bootcamp exclusivamente para navegar las clases existentes en el JDK... asi que lo reduciremos a dar algunos ejemplos de packages y su

funcion:

- **java.lang:** Clases esenciales como `Object`, `String`, y `System`.
- **java.util:** Estructuras de colecciones, como `List`, `Set`, `Map`, y clases de utilidad como `Collections`.
- **java.io:** Clases de entrada y salida para manejar flujos y archivos.
- **java.nio:** Nuevo I/O para operaciones de entrada/salida escalables y no bloqueantes.
- **java.net:** Clases de redes para trabajar con URL, sockets, etc.
- **java.time:** API de fecha y hora introducida en Java 8.
- **java.util.concurrent:** Utilidades de concurrencia para multithread
- **java.sql:** JDBC para conectividad con bases de datos.

Ejemplos de algunas clases en esos packages:

- **java.lang.Object:** La clase raíz para todas las clases en Java.
- **java.lang.String:** Representa una secuencia de caracteres.
- **java.util.ArrayList:** Implementación de un array dinámico de la interfaz List.
- **java.util.HashMap:** Implementa la interfaz Map utilizando una tabla de hash.
- **java.util.Scanner:** Permite la lectura de entrada de varios tipos desde diferentes fuentes.
- **java.io.File:** Representa la ruta de un archivo o directorio.
- **java.io.BufferedReader** y **java.io.InputStreamReader:** Utilizados para una lectura eficiente de texto desde flujos de entrada.
- **java.nio.file.Path** y **java.nio.file.Files:** Clases para la manipulación de archivos y directorios.
- **java.net.URL** y **java.net.HttpURLConnection:** Utilizados para el manejo de URL y conexiones HTTP.
- **java.time.LocalDate** y **java.time.LocalDateTime:** Clases para la manipulación de fechas y horas.
- **java.util.concurrent.ExecutorService** y **java.util.concurrent.Future:** Para programación concurrente y tareas asíncronas.
- **java.sql.Connection**, **java.sql.Statement** y **java.sql.ResultSet:** Clases centrales para la conectividad de bases de datos JDBC.

Ejemplos de codigo (snippets) seran incluidos en los recursos de la clase.

Manejo de excepciones

En Java, una excepción es un evento inesperado que ocurre durante la ejecución de un programa y puede interrumpir el flujo normal de ejecución. Las excepciones permiten manejar situaciones inesperadas de manera controlada.

Tipos de Excepciones

Java clasifica las excepciones en dos categorías principales:

- Excepciones Verificadas (Checked): Son aquellas que el compilador obliga a gestionar mediante el uso de bloques try-catch o mediante la declaración de que el método puede lanzar estas excepciones. Todas estas excepciones extienden (heredan) de `Exception` directamente. Sin un manejo apropiado, nuestro codigo no podra compilar.
- Excepciones No Verificadas (Unchecked): Son excepciones que el compilador no requiere manejar explícitamente. Incluyen errores del programador (por ejemplo, división por cero) y problemas en tiempo de ejecución (por ejemplo, acceso a un índice fuera de los límites). Todas estas excepciones heredan de `RuntimeException`.

En ambos casos, las excepciones heredan de `Throwable`. Existen tambien los `Errors`, este tipo son clases de algo fatal de lo cual el programa no se puede recuperar, usualmente algo que afecto su ejecucion y lo obligo a cerrar abruptamente. Usualmente son lanzados por la JVM para indicar un error en la programacion.

Bloques try-catch

Para manejar excepciones, se utilizan bloques try-catch. El código que podría lanzar una excepción se coloca dentro del bloque try, y las acciones a realizar en caso de que ocurra la excepción se especifican en el bloque catch.

```
try {
    // Código que podría lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Acciones a realizar si se captura la excepción
} finally {
    // Opcional: Bloque que se ejecuta siempre, independientemente de si se lanza o no una e
}
```

El bloque finally es opcional y se utiliza para definir código que se ejecutará siempre, ya sea que se lance o no una excepcion.

Existe en Java una interfaz para indicar cuando un objeto o clase adquiere un recurso del sistema y requiere ser cerrado, esta interfaz es `AutoClosable`, similar, existe la interfaz `Closable`. Cuando una clase implementa el metodo `close` presente en estas interfaces, puede usarse un mecanismo conocido como *try-with-resources*

Previo a java 8, era tedioso escribir estos bloques y dependiendo la clase era el momento en que debias llamar al metodo `close`:

```
SomeClosableClass variable;
try {
    variable = new SomeClosableClass(...);
} catch (SomeException ex){
    // Exception handling
} finally {
```

```
  variable.close()
}
```

Desde Java 8 ya no es necesario llamar al metodo `close`:

```
try(SomeClosableClass variable = new SomeClosableClass(...)) {

} catch (SomeException ex) {

}
```

Existen casos donde no sabremos como contener la excepcion o donde necesitamos que algun otra clase se encarge del manejo, en los casos de `RuntimeException` no es necesario hacer nada adicional, aunque no tendremos manera de evitar la propagacion de la excepcion ni de indicar que sucedio hasta que agreguemos en try-catch. Codigo nuevo que invoque nuestro codigo con excepcion no sabra que algo sale male hasta revisar los mensajes del programa.

Con las **Checked** excepcions es distinto... cualquier metodo que pueda generar una excepcion **Checked** debe explicitamente manejar la excepcion en un try-catch o agregar a su firma la clausula `throws`:

```
public void someFailingMethod() throws SomeCheckedException {

}
```

La mayor desventaja de esto es que con el pasar del tiempo, si no agregamos un try-catch veremos como muchos metodos empiezan a requerir `throws` para poder compilar el programa.

La recomendacion general es: Prefiere usar ***Unchecked*** excepcions cuando sea posible y trata de lidiar con la excepcion lo mas pronto posible.

En codigo mas moderno, existen alternativas a usar excepciones, sin embargo es extremedamente comun encontrarnos con estas excepciones en mucho codigo.

Colecciones

Las colecciones de Java representan un elemento fundamental de la ciencias de la computacion: Estructuras de datos. Exiten una multitud enorme de estructuras de datos, por eso mismo nos enfocaremos en 3 especificamente por ahora (pero puedes navegar mas en la documentacion oficial): List, Set y Map.

Las colecciones por lo general son genericas, lo que quiere decir que funcionan como contenedores de otro tipo de dato...

Un parentesis rapido sobre generics

Los genéricos son una característica que permite a los desarrolladores crear clases, interfaces y métodos que pueden trabajar con tipos de datos específicos

sin comprometer la seguridad de tipo. En lugar de definir un tipo específico al diseñar una clase o método, se utiliza un tipo genérico que se determina en tiempo de compilación.

La principal ventaja de los genéricos es proporcionar flexibilidad y reutilización del código, permitiendo que las clases y métodos trabajen con diferentes tipos de datos sin necesidad de escribir implementaciones específicas para cada tipo.

```
public class MyBox<T> {
    private T content;

    public T getContent() {
        return content;
    }

    public void setContent(T freshContent) {
        this.content = freshContent;
    }
}

///////

private void somewhereElse() {
     MyBox<String> myUser = new MyBox<String>();
     myUser.setContent("Sier");

     MyBox<URL> myWeb = new MyBox<URL>();
     myWeb.setContent(new URL("https://github.com/sierisimo"));
}
```

Los genericos pueden definir limites (bounds), para indicar que unicamente genericos con ciertas características pueden ser usados, para esto debemos usar `extends` o `super` al definir un generico.

```
// Se considera un Upper bound (limite superior) porque decimos que unicamente podemos trab
class Box<T extends Number> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void printInfo() {
        System.out.println("Type: " + content.getClass().getName());
```

```
        System.out.println("Content: " + content);
    }
}

// Se dice que es un Lower Bound porque usamos super para indicar que podemos operar con cu
// Esto permite usar Number, Integer u Object.
class DataProcessor {
    public static void processBoxes(Box<? super Integer> box) {
        // Perform operations with the box
        System.out.println("Processing box with content: " + box.getContent());
    }
}

public class Main {
    public static void main(String[] args) {
        // Ejemplo de Upper Bound
        Box<Integer> integerBox = new Box<>(42);
        integerBox.printInfo();

        // Ejemplo de Lower Bound
        Box<Number> numberBox = new Box<>(3);
        DataProcessor.processBoxes(numberBox);
    }
}
```

Volviendo a las colecciones

List Las listas son similares a los arreglos, son colecciones de datos secuenciales y que se encargan de almacenar elementos de un mismo tipo. Existen multiples implementaciones ya que `List` en Java es una interfaz generica.

Los metodos de `List` son mayormente usados para acceder e insertar elementos dentro de la lista, la mayor ventaja que aportan comparada con Arreglos es que son dinamicas, es decir, no tienen un tamano inicial definido y pueden crecer a demanda.

Las dos implementaciones mas populares de `List` son: `ArrayList` y `LinkedList`, la diferencia principal entre ambas es su funcionamiento interno. `ArrayList` usa un arreglo interno para almacenar los datos que tiene y conforme crece o decrece, recrea el arreglo donde almacena sus datos.

`LinkedList` hace algo diferente, tiene una estructura interna que se llama `Node`, la cual usa para registrar cada elemento en la lista y hacer que entre estos nodos "se apunten". De esa manera las inserciones y borrados son mas eficientes.

Set Un `Set` es una estructura en la cual, todos los elementos contenidos son unicos. Si en un `Set` nosotros insertamos un elemento ya existente, el `Set` no se

modifica, mantiene los mismos elementos.

La ventaja de un `Set` es que nos permite rapdidamente diferenciar elementos y las operaciones en el mismo son bastante rapidas, la operacione mas usual es la busqueda, en un `Set` es ridiculamente rapida.

Las clases mas comunes de `Set` son `LinkedHashSet` y `TreeSet`, en este caso la diferencia de como operan internamente solo es reelevante a gran escala.

Map Un `Map` es una estructura que guarda sus elementos en la forma `llave -> valor`. En otros lenguajes se le conoce como diccionario. `Map` es una interfaz generica y usa un tipo para la llave (`Key`) y otro tipo para el valor: `Map<String, User>`, cuando una Key no tiene un valor asignado, el valor es `null`. Asignar un valor a una Key previamente asignada solo sobreescribe el valor por el mas recientemente asignado.

La implementación mas comun de `Map` es `HashMap`, tambien es una de las mas simples de usar, sin embargo puede que no sea la mas adecuada para todos los casos.

Mas colecciones Hay muchas, muchas mas colecciones que podriamos pasar horas revisando, pero por cuestion de tiempo, dejaremos para investigar. Se pueden encontrar algunas en la documentacion oficial de Oracle y adicionalmente algunos frameworks y librerias ofrecen los suyos (Guava, Eclipse).

Algunas de las mas populares a revisar: `Vector`, `ConcurrentHashMap`, `Stack`, `Queue`, `Dequeue`, tratar de listarlas todas y explicarlas requeriria un curso por separado.

API de Fecha y hora

La API de tiempo en Java proporciona clases para manejar operaciones relacionadas con fechas y horas. Antes de Java 8, la manipulación de fechas se basaba en las clases `Date` y `Calendar`, que eran propensas a errores y carecían de funcionalidades modernas. A partir de Java 8, se introdujo una nueva API de tiempo más consistente y fácil de usar.

Previo a Java 8:

```java
// Java antes de Java 8
Date date = new Date();
Calendar calendar = Calendar.getInstance();
int year = calendar.get(Calendar.YEAR);

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
String formattedDate = sdf.format(date);
```

Java 8 y Posterior:

```java
// Java 8 y posterior
LocalDate currentDate = LocalDate.now();
int year = currentDate.getYear();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
String formattedDate = currentDate.format(formatter);
```

Con Java 8 y versiones posteriores, se introdujo la API de tiempo (java.time) que incluye clases como LocalDate, LocalTime, LocalDateTime, etc.:

```java
// Java 8 y posterior
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plusDays(1);

LocalTime currentTime = LocalTime.now();
LocalTime newTime = currentTime.plusHours(2);

LocalDateTime currentDateTime = LocalDateTime.now();
LocalDateTime newDateTime = currentDateTime.plusDays(1).minusHours(3);

Period period = Period.between(today, tomorrow);
Duration duration = Duration.between(currentTime, newTime);
```

La API de tiempo proporciona una manipulación de fechas y horas más segura y fácil de entender. Además, se introdujeron clases como Period y Duration para manejar diferencias entre fechas y horas de manera más efectiva. La interfaz DateTimeFormatter facilita el formateo y análisis de fechas y horas según patrones específicos.

Streams y expresiones lambda

Lambdas

Las expresiones lambda en Java introducidas en Java 8 permiten tratar las funciones como objetos. Esto simplifica la escritura de código más conciso y legible, especialmente en operaciones que implican el uso de interfaces funcionales.

```java
// Ejemplo de expresión lambda
List<String> listaNombres = Arrays.asList("Juan", "María", "Carlos");

// Antes de Java 8
Collections.sort(listaNombres, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
```

```
    }
});

// Con expresión lambda en Java 8
Collections.sort(listaNombres, (s1, s2) -> s1.compareTo(s2));
```

Streams en Java

Los streams en Java proporcionan una forma elegante y funcional de realizar operaciones en colecciones de datos. Permiten operaciones en elementos de manera secuencial o paralela, lo que puede mejorar significativamente el rendimiento en ciertos casos.

```
// Ejemplo de Stream en Java
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

// Operación de filtrado y mapeo
int sumaCuadradosPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .reduce(0, Integer::sum);

System.out.println(sumaCuadradosPares);  // Salida: 20
```

En este ejemplo, el stream filtra los números pares, luego aplica una función de mapeo para obtener el cuadrado de cada número y finalmente reduce los resultados sumándolos.

```
List<String> listaPalabras = Arrays.asList("Java", "es", "poderoso", "y", "elegante");

long cantidadMayusculas = listaPalabras.stream()
    .filter(palabra -> Character.isUpperCase(palabra.charAt(0)))
    .count();

System.out.println(cantidadMayusculas);  // Salida: 1
```

En este caso, se utiliza una expresión lambda dentro de un stream para contar las palabras que comienzan con mayúscula.

Debemos entender que los `Stream` son genericos y por lo tanto pueden convertir los datos que contienen sin problemas, dependera mayormente del tipo de operador que usemos para poder determinar el tipo resultante. En algunos casos, para poder obtener un cierto tipo final, debemos usar el operador `collect()`.

Conclusion del modulo 4

El JDK es inmeso en la cantidad de clases y utilidades que incluye, es importante que tratemos de identificar que tipo de operacion o accion queremos realizar para poder identificar correctamente que clase podemos usar dentro del JDK que nos facilite operar.

En el siguiente modulo veremos como ha ido evolucionando Java en cada version y que elementos se han ido agregando para hacer al lenguaje mas moderno.