Parte 1: Investigación sobre la Inmutabilidad en Maps

Conceptos Básicos

Inmutabilidad en Maps en Java: Un `Map` inmutable en Java es una colección que no puede ser modificada una vez que ha sido creada. Esto significa que no puedes agregar, eliminar o cambiar pares clave-valor en un `Map` inmutable después de su creación. La inmutabilidad es una propiedad importante en programación funcional, donde los datos no cambian su estado una vez creados

Creación de Maps Inmutables: Java proporciona varias formas de crear un `Map` inmutable:

1. **`Collections.unmodifiableMap(Map<K, V>)`:** Esta es una de las maneras más comunes de crear un `Map` inmutable. Este método toma un `Map` mutable y devuelve una vista inmutable de él. Cualquier intento de modificar el `Map` resultará en una `UnsupportedOperationException`.

   ```
   Map<String, Integer> mutableMap = new HashMap<>();
   mutableMap.put("Alice", 30);
   mutableMap.put("Bob", 25);

   Map<String, Integer> immutableMap =
   Collections.unmodifiableMap(mutableMap);
   ```

2. **`Map.of(...)`:** Introducido en Java 9, `Map.of` es una forma concisa de crear un `Map` inmutable con un número fijo de pares clave-valor. Este método es útil cuando se necesita un `Map` pequeño y constante.

   ```
   Map<String, Integer> immutableMap = Map.of("Alice", 30, "Bob", 25);
   ```

Ventajas de los Maps Inmutables

Seguridad y Concurrencia: Los Maps inmutables son naturalmente seguros para su uso en entornos concurrentes, ya que múltiples hilos pueden acceder a ellos sin necesidad de sincronización. Esto reduce el riesgo de errores relacionados con la concurrencia.

Previsibilidad y Simplicidad: Al garantizar que un `Map` no cambiará después de ser creado, los programas se vuelven más predecibles y fáciles de razonar. No hay

necesidad de rastrear cambios en el estado de los datos, lo que simplifica el análisis y la depuración.

Uso en Programación Funcional: En paradigmas de programación funcional, la inmutabilidad es fundamental. Permite que las funciones sean puras (sin efectos secundarios), lo que facilita la composición y reutilización del código.

Protección de la Integridad de los Datos: Los Maps inmutables protegen los datos de modificaciones accidentales o maliciosas, lo que es crucial en aplicaciones donde la integridad de los datos es prioritaria, como en sistemas financieros o de control.

Documentación y Fuentes

- Documentación Oficial de Java: Java SE Documentation
- Blog de Oracle sobre Colecciones Inmutables: Immutable Collections in Java
- Guía de Baeldung sobre Inmutabilidad: Immutable Maps in Java

Parte 2: Aplicación Práctica

1. Implementación de Map Inmutable (Código)

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class ImmutableMapExample {
    public static void main(String[] args) {
        // Crear un Map mutable y llenarlo con algunos pares clave-valor
        Map<String, Integer> mutableMap = new HashMap<>();
        mutableMap.put("Alice", 30);
        mutableMap.put("Bob", 25);

        // Convertir el Map en un Map inmutable usando Collections.unmodifiableMap
        Map<String, Integer> immutableMap = Collections.unmodifiableMap(mutableMap);

        // Mostrar el contenido del Map inmutable
        System.out.println("Contenido del Map inmutable: " + immutableMap);

        // Intentar modificar el Map inmutable y manejar la excepción
        try {
            immutableMap.put("Charlie", 35);
        } catch (UnsupportedOperationException e) {
            System.out.println("Error: No se puede modificar un Map inmutable.");
```

```
        }
    }
}
```

Explicación:

- **Creación de Map Mutable:** Se crea un `HashMap` que es mutable y se llena con algunos pares clave-valor.
- **Conversión a Map Inmutable:** Usamos `Collections.unmodifiableMap` para crear una vista inmutable del `Map`.
- **Manejo de Excepciones:** Al intentar modificar el `Map` inmutable, se lanza una `UnsupportedOperationException`, que es manejada en el bloque `try-catch`.

2. Ejemplo de Uso en Aplicaciones (Clases)

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

class Employee {
    private int id;
    private String name;
    private String position;

    public Employee(int id, String name, String position) {
        this.id = id;
        this.name = name;
        this.position = position;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", position='" + position + '\'' +
                '}';
    }
}

class Company {
    private Map<Integer, Employee> employeeMap = new HashMap<>();

    // Método para agregar empleados a un Map mutable
    public void addEmployee(Employee employee) {
```

```
        employeeMap.put(employee.getId(), employee);
    }

    // Método para convertir el Map en un Map inmutable y mostrar el contenido
    public Map<Integer, Employee> getImmutableEmployeeMap() {
        return Collections.unmodifiableMap(employeeMap);
    }

    // Intentar modificar el Map inmutable y manejar excepciones
    public void attemptModification() {
        Map<Integer, Employee> immutableMap = getImmutableEmployeeMap();
        try {
            immutableMap.put(4, new Employee(4, "David", "Designer"));
        } catch (UnsupportedOperationException e) {
            System.out.println("Error: No se puede modificar el Map inmutable de
empleados.");
        }
    }

    public void displayEmployees() {
        employeeMap.forEach((id, employee) -> System.out.println(employee));
    }
}
```

Clase Principal:

```
public class Main {

    public static void main(String[] args) {

        Company company = new Company();


        // Agregar empleados

        company.addEmployee(new Employee(1, "Alice", "Developer"));

        company.addEmployee(new Employee(2, "Bob", "Manager"));

        company.addEmployee(new Employee(3, "Charlie", "Analyst"));


        // Mostrar empleados antes de convertir el Map a inmutable
```

```
        System.out.println("Empleados antes de la inmutabilidad:");
        company.displayEmployees();

        // Convertir a Map inmutable y mostrar el contenido
        System.out.println("\nContenido del Map inmutable:");
        Map<Integer, Employee> immutableMap =
company.getImmutableEmployeeMap();
        immutableMap.forEach((id, employee) -> System.out.println(employee));

        // Intentar modificar el Map inmutable
        company.attemptModification();

        // Mostrar empleados después de intentar modificar el Map inmutable
        System.out.println("\nEmpleados después de intentar modificar el Map
inmutable:");
        company.displayEmployees();
    }
}
```

Explicación:

- **Clase `Employee`:** Define un empleado con atributos `id`, `name`, y `position`.
- **Clase `Company`:** Mantiene un `Map` mutable de empleados y proporciona métodos para agregar empleados, convertir el `Map` a inmutable, y manejar intentos de modificación.
- **Manejo de Excepciones:** Similar al ejemplo anterior, cualquier intento de modificar el `Map` inmutable es interceptado y gestionado.

Pruebas y Documentación

Pruebas:

- Se prueban varios escenarios para asegurar que el `Map` inmutable se comporte como se espera y que las excepciones se manejen correctamente.

Documentación:

- Se ha documentado el código con comentarios explicativos para facilitar su comprensión.

Conclusión: Este proyecto ha demostrado cómo y por qué se utilizan Maps inmutables en Java. La inmutabilidad ofrece ventajas claras en términos de seguridad, previsibilidad, y simplicidad, lo que la hace ideal para su uso en aplicaciones donde la consistencia de los datos es crucial.