

ACTIVIDAD 6.2 - Introspección y reflexión

Investigar sobre la característica de “Reflexión” que ofrece JAVA

▼ ¿Qué son la reflexión e introspección en Java?

Introspección: es la capacidad de un programa de **examinar** la estructura de un objeto o clase en tiempo de ejecución (qué atributos, métodos, constructores o anotaciones tiene) sin modificarla. En Java, el término se asocia sobre todo a los **JavaBeans** (`java.beans`), donde se descubren las *propiedades* de un bean a partir de sus getters/setters.

Reflexión: es una capacidad más amplia: además de examinar, permite **manipular** clases y objetos en tiempo de ejecución: instanciar objetos, invocar métodos, leer o modificar atributos (incluso privados) sin conocerlos en tiempo de compilación.

Paquetes:

- `java.lang.reflect` (Reflexión: `Method`, `Field`, `Constructor`, `Modifier`, `Array`...) junto con `java.lang.Class`.
- `java.beans` (Introspección: `Introspector`, `BeanInfo`, `PropertyDescriptor`).

▼ Métodos principales de Introspección

- `Introspector.getBeanInfo(Class<?>)`: obtiene la información del bean.
- `BeanInfo.getPropertyDescriptors()`: devuelve las propiedades.
- `BeanInfo.getMethodDescriptors()`: devuelve los métodos.
- `PropertyDescriptor.getName()`, `getPropertyType()`.
- `PropertyDescriptor.getReadMethod()` / `getWriteMethod()`: el getter y el setter de la propiedad.

También se suelen incluir los métodos "de consulta" de `Class`: `getName()`, `getSimpleName()`, `getSuperclass()`, `getInterfaces()`, `getDeclaredFields()`, `getDeclaredMethods()`, `getDeclaredConstructors()`, `getAnnotations()`.

▼ Métodos principales de Reflexión

Clase	Métodos clave
`Class`	`forName(String)`, `newInstance()` (obsoleto), `getDeclaredConstructor(...)`, `getMethod(...)`, `getDeclaredField(...)`
`Constructor`	`newInstance(Object... args)`
`Method`	`invoke(Object obj, Object... args)`, `getReturnType()`, `getParameterTypes()`
`Field`	`get(obj)`, `set(obj, valor)`, `getType()`
`AccessibleObject`	`setAccessible(true)` (salta las restricciones `private`)
`Modifier`	`isPublic()`, `isStatic()`, `isPrivate()`...

▼ Ejemplo

```
import java.lang.reflect.*;

class Persona {
    private String nombre = "Ana";
    public Persona() {}
    public void saludar(String otro) {
        System.out.println(nombre + " saluda a " + otro);
    }
}

public class Ejemplo {
    public static void main(String[] args) throws Exception {
        // Obtener la clase
        Class<?> c = Class.forName("Persona");

        // Introspección: listar métodos
        for (Method m : c.getDeclaredMethods()) {
            System.out.println("Método: " + m.getName());
        }

        // Reflexión: crear objeto, invocar método y leer campo privado
        Object obj = c.getDeclaredConstructor().newInstance();

        Method saludar = c.getMethod("saludar", String.class);
        saludar.invoke(obj, "Luis");
```

```
        Field f = c.getDeclaredField("nombre");
        f.setAccessible(true);
        System.out.println("Valor del campo: " + f.get(obj));
    }
}
```

▼ Crear un objeto a partir del nombre de una clase como `String`

Se usa `Class.forName` con el nombre completo (paquete incluido) y después el constructor:

java

```
String nombreClase = "com.miapp.modelo.Cliente";Class<?> clase = Class.forName(nombreClase);Object objeto = clase.getDeclaredConstructor().newInstance();
```

Si el constructor tiene parámetros:

java

```
Constructor<?> ctor = clase.getDeclaredConstructor(String.class, int.class);Object objeto = ctor.newInstance("Ana", 30);
```

Hay que gestionar excepciones como `ClassNotFoundException`, `NoSuchMethodException`, `InstantiationException`, `IllegalAccessException` e `InvocationTargetException`.

▼ ¿Se puede invocar un método sin conocerlo en compilación?

Sí. Se obtiene el objeto `Method` por su nombre (como `String`) y los tipos de sus parámetros, y se invoca con `invoke`:

java

```
String nombreMetodo = "calcularTotal";   // podría venir de un fichero o del usuarioMethod m = objeto.getClass().getMethod(nombreMetodo, double.class);Object resultado = m.invoke(objeto, 21.0);
```

Si el método es privado, se usa `getDeclaredMethod(...)` y `m.setAccessible(true)`. Para métodos estáticos, el primer argumento de `invoke` es `null`.

▼ Riesgos y desventajas de abusar de la reflexión

- **Rendimiento:** es más lenta que la llamada directa, y la JVM no puede optimizar igual.
- **Pérdida de seguridad en tiempo de compilación:** los errores (método inexistente, tipos incorrectos) aparecen en ejecución como excepciones, no al compilar.

- **Rotura del encapsulamiento:** `setAccessible(true)` permite acceder a miembros privados, lo que viola el diseño de la clase.
- **Mantenibilidad:** el código es más difícil de leer, depurar y refactorizar (renombrar un método con el IDE no actualiza los `String` que lo referencian).
- **Problemas de seguridad:** si los nombres de clase o método vienen de entrada externa, se puede ejecutar código no previsto.
- **Compatibilidad con módulos (Java 9+):** el sistema de módulos restringe el acceso reflexivo a paquetes no abiertos (`opens`).
- **Compilación nativa (GraalVM):** requiere configuración extra, porque la reflexión no se puede resolver de forma estática.

▼ Tres proyectos populares en Acceso a Datos

1. **Hibernate (JPA):** mediante reflexión lee las anotaciones (`@Entity`, `@Id`, `@Column`) de tus clases, instancia las entidades y rellena sus atributos con los resultados de la consulta SQL, sin que tú escribas ese código. También genera *proxies* para la carga perezosa (*lazy loading*).
2. **Spring Data / Spring Framework (Spring JDBC, Spring Data JPA):** usa reflexión para la inyección de dependencias (`@Autowired`), para implementar en tiempo de ejecución los repositorios a partir de interfaces (`JpaRepository`) y para mapear filas a objetos (`BeanPropertyRowMapper`, que usa introspección de propiedades).
3. **Jackson / Gson (serialización JSON) y MyBatis:** Jackson y Gson inspeccionan los atributos o getters/setters de un objeto para convertirlo a JSON y viceversa, algo habitual en APIs que acceden a datos. MyBatis también utiliza reflexión para mapear resultados SQL a objetos Java mediante sus *mappers*.