21-Serializable Interface

Author: Vincent Lau

Learning Objectives

> Understand the reasons for implementing Serializable Interface
>
> What scenario should we implement Serializable and what will happen if we don't make it?
>
> What are the types of Incompatible Changes in Serialization & Deserialization?
>
> Do we need to implement Serializable Interface in DTO & Entity of Spring Boot Development?

Introduction

The `Serializable` interface in Java is a marker interface, meaning it doesn't declare any methods or fields. Instead, it's used to indicate to the Java runtime that a class is serializable, i.e., its instances can be converted into a binary stream and then reconstructed. This is useful for saving objects to files, transmitting them over a network, or caching them.

Key Features

The `Serializable` interface is located in the `java.io` package, and it's used as follows:

```java
import java.io.Serializable;

public class MyClass implements Serializable {
    // ... class definition
}
```

Here are some key points about the `Serializable` interface:

1. Marker Interface

It doesn't require you to implement any methods; it's simply a marker to indicate that instances of the class can be serialized.

2. Serialization

When a class implements `Serializable`, its instances can be converted into a stream of bytes, which can then be saved to a file, sent over a network, or stored in a database.

3. serialVersionUID

When a class is serialized, a version identifier called `serialVersionUID` is associated with it. This identifier is a unique identifier generated by the Java compiler based on the class definition, including fields, methods, and their types. It's used to verify that the class being deserialized matches the one that was originally serialized.

4. serialVersionUID Generation

The `serialVersionUID` is typically generated automatically by the Java compiler. It's calculated based on the class's structure, and it provides version control for the class. If you don't provide a `serialVersionUID`, the Java compiler will generate one based on the class definition.

5. Version Control

By explicitly defining a `serialVersionUID` in your class, you can take control of the versioning process. This is important when you want to manage compatibility between different versions of your class.

SerialVersionUID

In a normal Java class that does not implement the `Serializable` interface, Java will not generate a `serialVersionUID`. The `serialVersionUID` is specific to classes that implement `Serializable` and is used for version control during the serialization and deserialization process.

If you have a class that doesn't implement `Serializable`, Java will not generate a `serialVersionUID` because the class is not intended to be serialized. The `serialVersionUID` is only relevant in the context of classes that are designed to be serialized, where it serves as a version control mechanism to ensure compatibility when the class structure changes.

If you have a class that implements `Serializable` and don't provide an explicit `serialVersionUID`, Java will automatically generate one based on the class's structure, including fields, methods, and their types. This auto-generated `serialVersionUID` is often sufficient for basic serialization needs.

InvalidClassException

Consider the following scenario.

This is an example to illustrate the problem that can occur when the `serialVersionUID` is not specified in a `Serializable` class. The `serialVersionUID` is used to ensure compatibility between the serialized object and the class structure. If it's not specified, the default `serialVersionUID` is generated based on various aspects of the class, and changes to the class can result in version conflicts.

In this example, we have a `MyData` class that implements `Serializable`, but it doesn't specify a `serialVersionUID`. If you run this code and serialize an object, everything works fine.

```
1  import java.io.*;
2
3  class MyData implements Serializable {
4      // Fields
5      private int value;
6
7      public MyData(int value) {
8          this.value = value;
9      }
10
11     public int getValue() {
12         return value;
```

```
    }
}

public class SerializationExample {

    public static void main(String[] args) {
        // Serialize an object
        try {
            MyData myData = new MyData(42);
            ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.ser"));
            out.writeObject(myData);
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize the object
        try {
            ObjectInputStream in = new ObjectInputStream(new
FileInputStream("data.ser"));
            MyData deserializedData = (MyData) in.readObject();
            in.close();
            System.out.println("Deserialized value: " +
deserializedData.getValue());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Now, let's **introduce a change** to the `MyData` class by adding a new field:

```
private String name;
```

After making this change, if you try to deserialize an object that was serialized before the change, you'll encounter a `java.io.InvalidClassException` because the class structure has changed. Here's the error you might see:

```
Exception in thread "main" java.io.InvalidClassException: MyData; local class
incompatible: stream classdesc serialVersionUID = -5258091914782955126, local
```

```
class serialVersionUID = 6145932995275128541
        at
java.base/java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:699)
        at
java.base/java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:2023)
        at
java.base/java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1881)
        at
java.base/java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2197)
        at
java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1679)
...
```

This error occurs because the deserialized object's version does not match the current class version.

To avoid this issue, you should specify a constant `serialVersionUID` in your `Serializable` class, and update it when you make structural changes to the class. This ensures that the deserialization process can verify the compatibility of the class versions and handle any changes appropriately.

Explicit Versioning

If you don't provide a `serialVersionUID` in your serializable class, Java will generate one based on the class's name, implemented interfaces, and other factors. However, this auto-generated value may not provide the version control you need to manage class compatibility over time. It's always a best practice to define your own `serialVersionUID` to ensure proper version control.

The `serialVersionUID` is a version control identifier for serialized objects. It's a `long` constant that you define in your serializable class. It's used to ensure that the serialized version of the class matches the deserialized version and to prevent compatibility issues when the class structure changes.

Here's how you define and provide a `serialVersionUID` in a serializable class:

```
import java.io.Serializable;

public class MySerializableClass implements Serializable {
    private static final long serialVersionUID = 2L; // Define your
serialVersionUID// ... rest of the class
}
```

In the example above, the `serialVersionUID` is defined as `123456789L`, but you can choose any long integer value for it. It's important to keep the `serialVersionUID` constant across versions of your class where the structure is compatible, and only update it when you make incompatible changes to your class.

Incompatible Changes

Incompatible changes are those that can potentially break the deserialization of previously serialized data. Such changes can lead to versioning issues, as the serialized data may no longer match the structure of the class. Here are some examples of incompatible changes:

1. Field Deletion

If you remove a field from the class, the deserialization process will fail when it encounters serialized data that includes the deleted field. The data structure no longer matches the class structure.

2. Field Type Change

Changing the data type of a field can break compatibility. For instance, if you change a field from an `int` to a `String`, the deserialization process will not be able to convert the serialized data to the new type.

3. Field Name Change

If you change the name of a field, the deserialization process will not find the field with the new name in the class, causing a failure.

4. Class Hierarchy Changes

Altering the inheritance hierarchy, such as changing the superclass or implementing new interfaces, can make the class incompatible with the previous version.

5. Method Signature Changes

Changes to method signatures that are used in the serialization or deserialization process can break compatibility.

6. Package Name Changes

Changing the package name of a class can make it incompatible with the previously serialized data.

In these cases, if the changes are made to the class and you still want to deserialize previously serialized data, **you should update the `serialVersionUID` to indicate that the class is now incompatible with the previous version**. This allows the deserialization process to handle the changes properly and avoid version conflicts.

Spring Boot Development

Data Transfer Object

The `Serializable` interface is NOT required for DTO classes when working with RESTful APIs.

The `Serializable` interface is primarily relevant when you want to serialize objects for purposes such as storing them in files, transmitting them over a network, or caching. In the context of RESTful APIs, data is typically exchanged in JSON or XML format, and you don't need to perform low-level serialization and deserialization of objects. The serialization and deserialization of objects to and from JSON or XML are handled by the framework, such as Spring Boot, without the need for the `Serializable` interface.

Here's an example of a DTO class used in a Spring Boot RESTful API:

```
public class MyDTO {
    private String name;
    private int age;

    // Constructors, getters, setters
}
```

You can use this DTO class as the response object in your controller's endpoint, and Spring Boot will automatically convert it to JSON or XML when sending the response. API consumers can receive the response in the same DTO class without any need for low-level serialization or the `Serializable` interface.

Entity

For entity classes that represent data stored in a database (e.g., MySQL, PostgreSQL) or data cached in systems like Redis, it is generally a good practice to implement the `Serializable` interface, even if they won't be directly serialized or deserialized in your application code. Implementing `Serializable` provides benefits related to serialization, and it can be relevant in certain situations:

1. **Compatibility:** Implementing `Serializable` helps ensure compatibility when working with different versions of your application. If you ever need to serialize or cache these entities, having the `Serializable` interface can prevent compatibility issues (i.e. InvalidClassException in the previous section).
2. **Caching:** If you plan to use caching mechanisms like Hibernate's second-level cache or distributed caches (e.g., Redis cache), these mechanisms may serialize and deserialize entities for efficient caching. Implementing `Serializable` ensures that entities can be safely cached.
3. **Future Use:** While you may not need serialization now, it provides flexibility for future use cases where you might want to serialize entities for purposes like distributed computing or data synchronization.

Here's how you might implement `Serializable` in an entity class:

```
1  import java.io.Serializable;
2  import javax.persistence.Entity;
3  import javax.persistence.Id;
4
5  @Entity
6  public class MyEntity implements Serializable {
7
8      private static final long serialVersionUID = 1L; // Define your
   serialVersionUID
9
10     @Id
11     private Long id;
12     private String name;
13
14     // Constructors, getters, setters, etc.
15 }
```

Summary

In summary, the `Serializable` interface is used to mark classes as serializable, and it enables objects of these classes to be converted into a binary stream. The `serialVersionUID` is an important aspect of serialization, and it's automatically generated or can be explicitly defined to control versioning and ensure compatibility between different versions of the class.